VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Barnesův-Hutův algoritmus pro řešení problému mnoha těles

Barnes-Hut algorithm for n-body problem

2019 Jakub Homola

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Zadání bakalářské práce

Student: **Jakub Homola**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 1103R031 Výpočetní matematika

Téma: Barnesův-Hutův algoritmus pro řešení problému mnoha těles
Barnes-Hut algorithm for n-body problem

Jazyk vypracování: čeština

Zásady pro vypracování:

Problém mnoha těles (n-body problem) se objevuje např. při zkoumání pohybu galaxií, hvězdokup nebo jiných vesmírných objektů, ale i při predikci chování skupiny atomů či molekul. K určení pohybu každého jednotlivého tělesa je třeba sečíst silové působení všech ostatních těles na toto těleso. Celková náročnost naivního přístupu je tedy kvadratická, což je pro systémy obsahující i desítky milionů objektů nepoužitelné. Existují ale i sofistikovanější algoritmy, které jsou schopny problém řešit s téměř lineární složitostí (čas řešení tedy roste jako N*log(N), kde N je počet těles), např. Barnesův-Hutův algoritmus. Ten je založen na hierarchickém dělení výpočetní oblasti na shluky/clustery částic a vytváření odpovídající stromové struktury. Jednotlivým shlukům je přiřazena celková hmotnost a těžiště. Při výpočtu působení sil na těleso se prochází strom a dostatečně vzdálené shluky jsou brány jako jeden objekt s daným těžištěm a hmotností – není tak třeba procházet každou částici vzdáleného clusteru samostatně.
Práce bude strukturovaná následovně:

1) úvod, popis problému, možnosti řešení,
2) popis Barnesova-Hutova algoritmu,
3) popis existujících kódů nebo vlastní implementace; možno doplnit také o popis optimalizace/paralelizace,
4) numerické experimenty, vizualizace řešení, možno i porovnat s existujícími kódy,
5) závěr.

Kromě samotného algoritmu se student seznámí i s numerickými metodami pro řešení obyčejných diferenciálních rovnic (např. s Eulerovou metodou). Matematickou stránku práce je možné doplnit také o analýzu chyby metody. Z implementačního hlediska by práce měla studenta obohatit o zkušenosti s vývojem vědeckého kódu nejlépe v C/C++. Je možné se zaměřit také na jeho optimalizaci a případně paralelizaci. Na téma lze v budoucnu navázat a věnovat se sofistikovanějším algoritmům, např. fast multipole method, který je označován za jeden z deseti nejvýznamnějších algoritmů 20. století.

Seznam doporučené odborné literatury:

Barnes, J.; Hut, P. A hierarchical O(N log N) force-calculation algorithm. Nature, vol. 324, 1986.
Vondrák, V.; Pospíšil, L. Numerické metody I. VŠB-TU Ostrava.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Michal Merta, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

prof. RNDr. Jiří Bouchala, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 30. dubna 2019 Homola .....

Děkuji Ing. Michalu Mertovi, Ph.D. za odborné vedení bakalářské práce, věnovanou práci a čas, konstruktivní kritiku, věcné připomínky, poskytnuté rady a přístup k výpočetním zdrojům.

Abstrakt

Problém mnoha těles spočívá v nalezení trajektorií několika těles (např. planet, hvězd), na něž působí pouze vzájemná gravitační síla. Tento problém je analyticky řešitelný pro nejvýše dvě tělesa, k predikci chování systému obsahujícího více těles je tedy nutné hledat řešení numericky. V této práci probereme několik numerických metod pro řešení počátečních úloh, které jsou k řešení tohoto problému nezbytné. Dále se zabýváme dvěma algoritmy pro výpočet vzájemných sil působících na tělesa – přímou metodou s časovou složitostí $\mathcal{O}(n^2)$ a zejména Barnesovým-Hutovým algoritmem, který má příznivější časovou složitost $\mathcal{O}(n \log n)$. V rámci této práce byl vytvořen řešič problému mnoha těles, který je implementován v jazyce C++ a paralelizován pomocí OpenMP. Efekivitu vytvořeného kódu demonstrujeme na několika numerických experimentech.

Klíčová slova: problém mnoha těles, částicová simulace, Barnesův-Hutův algoritmus, řešení počátečních úloh, C++, OpenMP, paralelizace

Abstract

The goal of an N-body problem is to find trajectories of a number of bodies (e.g. planets, stars) interacting only by their mutual gravitational forces. This problem is analytically solvable for at most two bodies. To predict the motion of a larger number of bodies, numerical methods have to be used. In this work we cover several numerical methods for solution of an initial value problem, which is necessary for finding the numerical solution of an N-body problem. Furthermore, we analyze two algorithms for calculating mutual forces acting on particles – direct method with $\mathcal{O}(n^2)$ time complexity and mainly the Barnes-Hut algorithm with more favorable $\mathcal{O}(n \log n)$ time complexity. A solver for the N-body problem is implemented in C++ and paralelized using OpenMP. The efficiency of the program is demonstrated on several numerical experiments.

Key words: N-body problem, particle simulation, Barnes-Hut algorithm, initial value problem, C++, OpenMP, parallelization

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

1 Úvod

Problém mnoha těles se objevuje například při predikci chování galaxií, hvězdokup nebo planetárních systémů. Jádrem problému je nalezení trajektorií těles, na které působí pouze vzájemná gravitační síla. K tomu je třeba numericky řešit soustavu diferenciálních rovnic a počítat síly působící na jednotlivá tělesa. Oběma těmto částem se v této práci podrobněji věnujeme.

Zmíněná soustava diferenciálních rovnic popisující pohyb částic v systému má analytické řešení pro nejvýše dvě částice. Pro větší počet částic je nutné tuto soustavu řešit s použitím numerických metod. V práci popisujeme Eulerovu metodu, metodu leapfrog a Rungeho-Kuttovy metody. Vybrané metody jsou aplikovány na problém mnoha těles.

Pro výpočet síly působící na těleso je přímočarým řešením sečtení silového působení všech ostatních těles. Tento přístup má však vysokou časovou složitost $\mathcal{O}(n^2)$. Existují ale sofistikovanější algoritmy výpočtu sil se složitostí $\mathcal{O}(n \log n)$, jedním z nichž je Barnesův-Hutův algoritmus. Sofistikovanějšími algoritmy, které mají složitost $\mathcal{O}(n)$, se v této práci zabývat nebudeme.

Barnesův-Hutův algoritmus rekurzivně rozdělí simulovaný prostor na krychlové buňky, kterým přiřadí jejich velikost, polohu a celkovou hmotnost. Při výpočtu síly působící na částici se pak vytvořená struktura rekurzivně prochází a pokud je buňka dostatečně malá a vzdálená, je pro účel výpočtu síly aproximována jedinou virtuální částicí. V opačném případě je výsledkem součet silových působení všech podbuněk příslušné buňky. Tím se za cenu mírného snížení přesnosti simulace zlepší časová složitost algoritmu.

Pro simulaci částicových systémů existuje několik volně dostupných softwarových řešení, například balíky StarLab [6] nebo Amuse [22]. V minulosti pro částicové simulace existovala řada hardwarových akcelerátorů pod označením GRAPE (gravity pipe) [19]. Tato zařízení měla hardware přímo uzpůsobený k výpočtu vzájemných sil přímou metodou, čímž bylo dosaženo značného navýšení výpočetního výkonu. V dnešní době se pro zrychlení výpočtů přešlo spíše k používání grafických karet. V této práci používáme výhradně vlastní implementaci v C++ optimalizovanou pro současné vícejádrové procesory. K paralelizaci ve sdílené paměti a vektorizaci využíváme standardu OpenMP.

Práce je členěna následovně. V kapitole 2 se od Newtonových zákonů doberene ke zmíněné soustavě diferenciálních rovnic. Kapitolu 3 věnujeme numerickým metodám řešení počátečních úloh. Zabýváme se zde Eulerovou metodou, metodou leapfrog a Rungeho-Kuttovými metodami. V kapitole 4 se podrobněji zabýváme oběma algoritmy výpočtu sil působících na částice, a to přímou metodou a zejména Barnesovým-Hutovým algoritmem. V kapitole 5 představíme vlastní implementaci programu pro řešení problému mnoha těles a detailněji se zabýváme paralelizací pomocí standardu OpenMP. Poslední kapitola 6 je věnována generování počátečních podmínek podle Plummerova modelu, měření chyb numerických metod a zejména testování efektivity a paralelní škálovatelnosti vytvořeného programu. V závěru práce shrneme dosažené výsledky.

2 Problém mnoha těles

2.1 Formulace problému

Mějme soustavu n hmotných bodů (částic) v prostoru $\mathbb{R}^3$, kde každý (i-tý) hmotný bod má konstantní kladnou hmotnost m_i, počáteční polohový vektor $\boldsymbol{r}_i^{(0)}$ a počáteční vektor rychlosti $\boldsymbol{v}_i^{(0)}$ v nějakém čase $t^{(0)}$. Cílem je dle Newtonova gravitačního zákona předpovědět vzájemné gravitační síly působící mezi hmotnými body a pomocí toho stanovit jejich trajektorie.

2.2 Postup řešení

Newtonův gravitační zákon říká, že gravitační síla, kterou j-tý hmotný bod působí na i-tý hmotný bod, je dána rovnicí

$$\boldsymbol{F}_{ij} = G\frac{m_i m_j}{\|\boldsymbol{r}_{ij}\|^3}\boldsymbol{r}_{ij} = G\frac{m_i m_j}{\|\boldsymbol{r}_j - \boldsymbol{r}_i\|^3}\left(\boldsymbol{r}_j - \boldsymbol{r}_i\right), \tag{2.1}$$

kde m_i, m_j a $\boldsymbol{r}_i$, $\boldsymbol{r}_j$ jsou po řadě hmotnosti a polohové vektory daných hmotných bodů a G je gravitační konstanta $G \approx 6,674 \cdot 10^{-11}\mathrm{N} \cdot \mathrm{kg}^{-2} \cdot \mathrm{m}^2$.

Celkovou sílu působící na danou částici získáme sečtením gravitačního působení všech ostatních částic

$$\boldsymbol{F}_i = \sum_{j=1, j\neq i}^{n} \boldsymbol{F}_{ij}. \tag{2.2}$$

Dále díky druhému Newtonovu pohybovému zákonu víme, že celková síla působící na částici je rovna součinu hmotnosti a zrychlení. Zrychlení lze také vyjádřit jako derivaci rychlosti nebo druhou derivaci polohy

$$\boldsymbol{F}_i = m_i\boldsymbol{a}_i = m_i\boldsymbol{v}_i' = m_i\boldsymbol{r}_i''. \tag{2.3}$$

Po zkombinování rovnic (2.1) – (2.3) a pokrácení m_i dostaneme diferenciální rovnici

$$\boldsymbol{r}_i'' = \sum_{j=1, j\neq i}^{n} \left(G\frac{m_j}{\|\boldsymbol{r}_j - \boldsymbol{r}_i\|^3}\left(\boldsymbol{r}_j - \boldsymbol{r}_i\right)\right), \tag{2.4}$$

kterou se řídí pohyb částic v daném systému. Tato rovnice vlastně představuje soustavu diferenciálních rovnic – každé i-té částici přísluší jedna takováto rovnice. Hledání řešení této soustavy diferenciálních rovnic pro neznámé funkce $\{\boldsymbol{r}_i(t)\}_{i=1}^n$, tj. hledání trajektorií částic v systému, je jádrem celého problému mnoha těles.

Definujme funkci $\boldsymbol{f}_i \colon \mathbb{R}^{3n} \to \mathbb{R}^3$, která představuje zrychlení působící na i-tou částici v závislosti na polohách všech částic, předpisem

$$\boldsymbol{f}_i(\boldsymbol{r}_1, \boldsymbol{r}_2, ..., \boldsymbol{r}_n) = \sum_{j=1, j\neq i}^{n} \left(G\frac{m_j}{\|\boldsymbol{r}_j - \boldsymbol{r}_i\|^3}\left(\boldsymbol{r}_j - \boldsymbol{r}_i\right)\right). \tag{2.5}$$

Rovnice (2.4) po dosazení (2.5) přejde v rovnici

$$\boldsymbol{r}_i'' = \boldsymbol{f}_i(\boldsymbol{r}_1, \boldsymbol{r}_2, ..., \boldsymbol{r}_n), \tag{2.6}$$

kterou můžeme zapsat vektorově jako

$$\boldsymbol{r}'' = \boldsymbol{f}(\boldsymbol{r}), \tag{2.7}$$

kde $\boldsymbol{r} = (\boldsymbol{r}_1, \boldsymbol{r}_2, ..., \boldsymbol{r}_n)$ je vektor poloh částic a $\boldsymbol{f}(\boldsymbol{r}) = (\boldsymbol{f}_1(\boldsymbol{r}), \boldsymbol{f}_2(\boldsymbol{r}), ..., \boldsymbol{f}_n(\boldsymbol{r}))$, je funkce, která počítá zrychlení částic.

Dále víme, že rychlost je derivací polohy, tj. $\boldsymbol{v} = \boldsymbol{r}'$. Tento vztah použijeme v rovnici (2.7) a dostaneme soustavu dvou diferenciálních rovnic prvního řádu

$$\begin{aligned} \boldsymbol{r}'(t) &= \boldsymbol{v}(t), \\ \boldsymbol{v}'(t) &= \boldsymbol{f}(\boldsymbol{r}(t)). \end{aligned} \tag{2.8}$$

Z předpokladů (viz podkapitolu 2.1) známe i počáteční podmínky

$$\begin{aligned} \boldsymbol{r}(t^{(0)}) &= \boldsymbol{r}^{(0)}, \\ \boldsymbol{v}(t^{(0)}) &= \boldsymbol{v}^{(0)}. \end{aligned} \tag{2.9}$$

Našim cílem je tedy vyřešit Cauchyho úlohu (2.8), (2.9) pro $t \in \langle t^{(0)}, t^{(0)} + T \rangle$, kde $T \in \mathbb{R}^+$ je délka časového intervalu, na kterém chceme znát řešení. Pro jednu a pro dvě částice v systému existuje obecné řešení [19], pro $n > 2$ se řešení musí hledat numericky.

3 Numerické řešení počátečních úloh

V této kapitole vycházíme z [12, 15, 20]. Mějme Cauchyho úlohu

$$\begin{aligned} y'(t) &= g(t, y(t)), \\ y(t^{(0)}) &= y^{(0)}, \end{aligned} \tag{3.1}$$

kterou chceme řešit na intervalu $\langle t^{(0)}, t^{(0)} + T\rangle$.

Všechny níže probírané metody numerického řešení počátečních úloh mají společné to, že interval, na kterém řešení hledáme, rozdělíme na několik podintervalů, v jejichž krajních bodech hledáme hodnoty neznámé funkce. Takovéto numerické řešení tedy nenajde hledanou funkci, ale pouze přibližné funkční hodnoty v několika bodech. Interval $\langle t^{(0)}, t^{(0)} + T\rangle$ tedy rozdělíme na m podintervalů $\langle t^{(k)}, t^{(k+1)}\rangle$ s ekvidistantními[1] krajními body $t^{(k)} = t^{(0)} + k\Delta t$, kde $\Delta t = \frac{T}{m}$. V těchto bodech hledáme přibližné hodnoty $y^{(k)} \approx y(t^{(k)})$.

Hodnoty $y^{(k)}$ jsou počítány s využitím jedné nebo více hodnot z předchozích kroků. Pokud pro výpočet $y^{(k)}$ používáme pouze jednu hodnotu (obvykle $y^{(k)}$ nebo $y^{(k-1)}$), je metoda *jednokroková*. Jestliže je pro výpočet použito K hodnot, mluvíme o *vícekrokové* nebo *K-krokové* metodě.

Pokud se v předpisu pro výpočet $y^{(k)}$ objevuje $y^{(k)}$ pouze na levé straně a na pravé straně nefiguruje, jde o *explicitní* metodu a spočtení $y^{(k)}$ je otázkou prostého vyhodnocení pravé strany předpisu. Pokud se ale $y^{(k)}$ na pravé straně vyskytuje, jedná se o *implicitní* metodu a pro výpočet $y^{(k)}$ je obecně třeba řešit nelineární rovnici, což přidává na složitosti. Implicitní metody však bývají stabilnější a v některých případech může být výhodné ji použít i přes zvýšenou složitost výpočtu jednoho kroku. To však není náš případ, budeme se tedy zabývat zejména explicitními metodami.

Protože se při řešení používají přibližné hodnoty a počítá se s aproximacemi, dopouštíme se při numerickém řešení počátečních úloh chyb. Těmi jsou zaokrouhlovací chyby, které nebudeme brát v úvahu, a chyby aproximace, které jsou dvojího druhu – lokální a globální diskretizační chyba.

Lokální diskretizační chyba $d^{(k)}$ je rozdíl mezi přibližně vypočtenou hodnotou a přesným řešením po jednom kroku,

$$d^{(k)} = y(t^{(k+1)}) - y^{(k+1)}, \tag{3.2}$$

přičemž předpokládáme přesnou znalost všech potřebných předchozích hodnot před $y^{(k+1)}$. Do lokální diskretizační chyby se tedy nezapočítávají žádné nepřesnosti z předchozích kroků, jde čistě o chybu, které se dopustíme při provedení jednoho kroku metody.

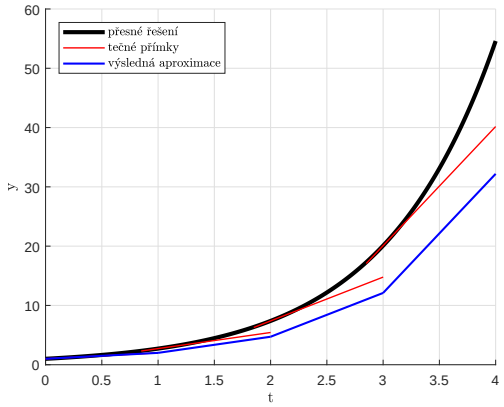
Globální diskretizační chyba vyjadřuje celkovou nepřesnost, o kterou se liší aproximované hodnoty od skutečných po nějakém čase T, neboli po $T/\Delta t$ krocích. Globální diskretizační chyba akumuluje lokální diskretizační chyby všech provedených kroků.

Řád metody je největší přirozené číslo p takové, že pro $\Delta t \to 0$ je lokální diskretizační chyba řádu Δt^{p+1}, tedy

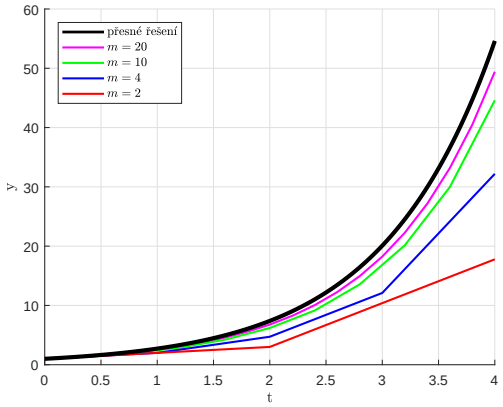
$$d^{(k)} = \mathcal{O}(\Delta t^{p+1}). \tag{3.3}$$

V následujících podkapitolách probereme několik numerických metod pro řešení počátečních úloh.

[1] délky podintervalů mohou být obecně různé, budeme se však zabývat pouze ekvidistantním dělením



Obrázek 3.1: Princip Eulerovy metody



Obrázek 3.2: Porovnání Eulerovy metody pro různé počty podintervalů

3.1 Eulerova metoda

Mějme zadánu Cauchyho úlohu (3.1), kterou řešíme na intervalu $\langle t^{(0)}, t^{(0)}+T\rangle$. Hledáme přibližné hodnoty $y^{(k)}$ aproximující funkční hodnoty $y(t^{(k)})$ na síti $m+1$ bodů $\{t^{(k)}\}_{k=0}^{m}$.

Mějme spočtenu přibližnou hodnotu hledané funkce v bodě $t^{(k)}$. Nyní chceme vyčíslit hodnotu v následujícím bodě $t^{(k+1)}$. Předpis pro ni odvodíme pomocí Taylorova polynomu. Neznámou funkci jím tedy nahradíme v bodě $t^{(k)}$,

$$y(t) = y(t^{(k)}) + (t - t^{(k)}) \cdot y'(t^{(k)}) + \frac{1}{2}(t - t^{(k)})^2 \cdot y''(t^{(k)}) + \cdots . \tag{3.4}$$

Na pravé straně ponecháme pouze první dva členy, hledanou funkci tedy v bodě $t^{(k)}$ nahrazujeme tečnou přímkou. Za derivaci funkce y dosadíme podle (3.1) funkci g,

$$y(t) \approx y(t^{(k)}) + (t - t^{(k)}) \cdot g(t^{(k)}, y(t^{(k)})). \tag{3.5}$$

Chceme znát předpis pro hodnotu v bodě $t^{(k+1)}$, do předchozí rovnice tedy dosadíme a upravíme,

$$y(t^{(k+1)}) \approx y(t^{(k)}) + (t^{(k+1)} - t^{(k)}) \cdot g(t^{(k)}, y(t^{(k)})) = y(t^{(k)}) + \Delta t \cdot g(t^{(k)}, y(t^{(k)})). \tag{3.6}$$

Nyní aproximujeme $y^{(k)} \approx y(t^{(k)})$, čímž dostaneme předpis explicitní (dopředné) Eulerovy metody

$$y^{(k+1)} = y^{(k)} + \Delta t \cdot g(t^{(k)}, y^{(k)}). \tag{3.7}$$

Jsme tedy schopni postupně spočítat přibližné hodnoty ve všech bodech $\{t^{(k)}\}_{k=0}^{m}$. Princip Eulerovy metody je znázorněn na obrázku 3.1. Na obrázku 3.2 jsou pro porovnání přesnosti vykresleny řešení jednoduché úlohy $y'(t) = e^t$, $y(0) = 1$ pro různé počty podintervalů.

Pokud bychom namísto derivace hledané funkce v bodě $t^{(k)}$ použili derivaci v bodě $t^{(k+1)}$, dostali bychom předpis implicitní (zpětné) Eulerovy metody

$$y^{(k+1)} = y^{(k)} + \Delta t \cdot g(t^{(k+1)}, y^{(k+1)}). \tag{3.8}$$

Odvoďme lokální diskretizační chybu a řád Eulerovy metody. Použijeme Taylorův rozvoj (3.4), do něhož dosadíme $t^{(k+1)}$ a upravíme

$$y(t^{(k+1)}) = y(t^{(k)}) + \Delta t \cdot y'(t^{(k)}) + \frac{1}{2}\Delta t^2 \cdot y''(t^{(k)}) + \frac{1}{6}\Delta t^3 \cdot y'''(t^{(k)}) + \cdots . \tag{3.9}$$

Spolu s (3.7) dosadíme do (3.2), přičemž víme, že $y^{(k)} = y(t^{(k)})$

$$\begin{aligned} d^{(k)} &= y(t^{(k+1)}) - y^{(k+1)} \\ &= -y^{(k)} - \Delta t \cdot g(t^{(k)}, y^{(k)}) + y(t^{(k)}) + \Delta t \cdot y'(t^{(k)}) + \frac{1}{2}\Delta t^2 \cdot y''(t^{(k)}) + \cdots \\ &= -\Delta t \cdot y'(t^{(k)}) + \Delta t \cdot y'(t^{(k)}) + \frac{1}{2}\Delta t^2 \cdot y''(t^{(k)}) + \frac{1}{6}\Delta t^3 \cdot y'''(t^{(k)}) + \cdots \\ &= \frac{1}{2}\Delta t^2 \cdot y''(t^{(k)}) + \frac{1}{6}\Delta t^3 \cdot y'''(t^{(k)}) + \cdots . \end{aligned} \tag{3.10}$$

Při $\Delta t \to 0$ člen s Δt^2 převládá nad ostatními, čímž zjišťujeme, že $d^{(k)} = \mathcal{O}(\Delta t^2)$. Eulerova metoda je tedy metoda prvního řádu. Podobně lze odvodit, že globální diskretizační chyba Eulerovy metody je v řádu $\mathcal{O}(\Delta t)$ [15].

Eulerova metoda je základní a nejjednodušší metodou numerického řešení počátečních úloh, je ale v porovnání s ostatními vcelku nepřesná. Pro $\Delta t \to 0$ však přibližné hodnoty $y^{(k)}$ konvergují ke skutečným $y(t^{(k)})$. Eulerova metoda je jednokrokovou metodou.

3.2 Metody Runge-Kutta

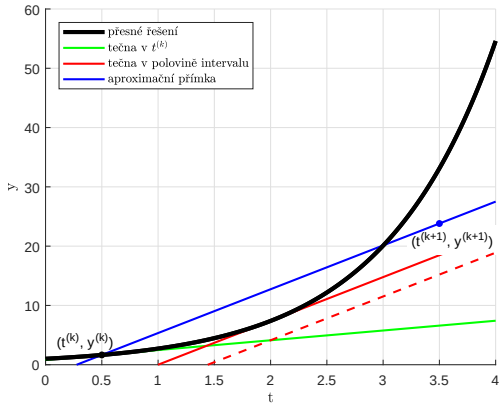
Metody Runge-Kutta jsou (stejně jako Eulerova metoda) jednokrokové metody, při výpočtu následující hodnoty $y^{(k+1)}$ je tedy třeba znát hodnotu pouze v předchozím kroku. Za cenu většího počtu vyhodnocení funkce g se však sníží diskretizační chyby a zvýší řád metody.

Princip je takový, že se pomocí odhadů derivací funkce y v několika bodech z intervalu $\langle t^{(k)}, t^{(k+1)} \rangle$ odhadne průměrná hodnota derivace na celém tomto intervalu. Funkci y na intervalu aproximujeme přímkou vycházející z bodu $(t^{(k)}, y^{(k)})$ se směrnicí rovnou odhadu průměrné derivace (průměrná derivace na intervalu je směrnicí přímky procházející funkčními hodnotami v krajních bodech intervalu). Hodnotou v následujícím kroku pak bude hodnota této přímky v následujícím bodě. Předpis pro hodnotu v následujícím kroku lze obecně zapsat jako

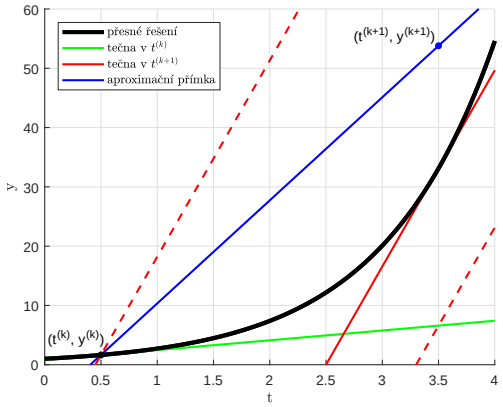
$$y^{(k+1)} = y^{(k)} + \Delta t \cdot \sum_{j=1}^{r} \alpha_j k_j, \tag{3.11}$$

kde α_j jsou váhy váženého průměru, k_j jsou odhady derivací a r je počet bodů, ve kterých budeme odhady derivací počítat. Hodnota k_j se obvykle počítá v závislosti na znalosti k_{j-1} a při jejím výpočtu je provedeno vyhodnocení funkce g. Obecně lze napsat

$$\begin{aligned} k_1 &= g(t^{(k)}, y^{(k)}), \\ &\vdots \\ k_j &= g(t^{(k)} + \lambda_j \Delta t, y^{(k)} + \mu_j \Delta t k_{j-1}), \end{aligned} \tag{3.12}$$



Obrázek 3.3: Modifikovaná Eulerova metoda



Obrázek 3.4: Heuného metoda

kde parametry λ_j a μ_j se volí v závislosti na konkrétní metodě. V následujících odstavcích je pro hlubší náhled uvedeno více Rungeho-Kuttových metod.

Příkladem Rungeho-Kuttovy metody pro $r = 2$ je modifikovaná Eulerova metoda. Volíme $\alpha_1 = 0$, $\alpha_2 = 1$ a

$$\begin{aligned} k_1 &= g(t^{(k)}, y^{(k)}), \\ k_2 &= g(t^{(k)} + \frac{\Delta t}{2}, y^{(k)} + \frac{\Delta t}{2} \cdot k_1). \end{aligned} \tag{3.13}$$

Předpis pro přibližnou hodnotu v následujícím kroku podle modifikované Eulerovy metody lze po úpravách zapsat jako

$$y^{(k+1)} = y^{(k)} + \Delta t \cdot g(t^{(k)} + \frac{\Delta t}{2}, y^{(k)} + \frac{\Delta t}{2} \cdot g(t^{(k)}, y^{(k)})). \tag{3.14}$$

Odhadem průměrné derivace funkce y na celém intervalu je hodnota derivace ve středu intervalu. Pro její výpočet, tedy pro vyčíslení funkce g uprostřed intervalu, ale potřebujeme znát i hodnotu funkce y v tomto bodě. Tu odhadneme pomocí standardní Eulerovy metody. Modifikovaná Eulerova metoda je metodou druhého řádu [15]. Je znázorněna na obrázku 3.3, na kterém si můžeme všimnout, že se dopouštíme menší chyby, než při provedení dvou kroků standardní dopředné Eulerovy metody (viz přerušovaná čára).

Dalším příkladem pro $r = 2$ je Heuného metoda, kde $\alpha_1 = \alpha_2 = \frac{1}{2}$ a

$$\begin{aligned} k_1 &= g(t^{(k)}, y^{(k)}), \\ k_2 &= g(t^{(k+1)}, y^{(k)} + \Delta t \cdot k_1). \end{aligned} \tag{3.15}$$

Směrnici aproximační přímky v tomto případě počítáme jako průměr derivací v krajních bodech intervalu $\langle t^{(k)}, t^{(k+1)} \rangle$, přičemž hodnotu funkce y nutnou pro vyčíslení její derivace v bodě $t^{(k+1)}$ odhadneme pomocí standardní Eulerovy metody. Heuného metoda je řádu 2 [15]. Je vyobrazena na obrázku 3.4, ze kterého je patrné, že dochází k menší chybě než u standardní Eulerovy metody (dopředné i zpětné, viz přerušované čáry).

Nejpoužívanější a nejoblíbenější je Rungeho-Kuttova metoda čtvrtého řádu s předpisem

$$y^{(k+1)} = y^{(k)} + \Delta t \cdot \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}, \tag{3.16}$$

kde

$$\begin{aligned} k_1 &= g(t^{(k)}, y^{(k)}), \\ k_2 &= g(t^{(k)} + \frac{\Delta t}{2}, y^{(k)} + \frac{\Delta t}{2} k_1), \\ k_3 &= g(t^{(k)} + \frac{\Delta t}{2}, y^{(k)} + \frac{\Delta t}{2} k_2), \\ k_4 &= g(t^{(k)} + \Delta t, y^{(k)} + \Delta t \cdot k_3). \end{aligned} \tag{3.17}$$

3.3 Metoda leapfrog

Metoda leapfrog [21] se na rozdíl od ostatních zmíněných metod vztahuje na diferenciální rovnice

$$\begin{aligned} r'(t) &= v(t), \\ v'(t) &= g(r(t)) \end{aligned} \tag{3.18}$$

s počátečními podmínkami

$$\begin{aligned} r(t^{(0)}) &= r^{(0)}, \\ v(t^{(0)}) &= v^{(0)}. \end{aligned} \tag{3.19}$$

Leapfrog metoda se nejhojněji používá k řešení počátečních úloh klasické mechaniky, r tedy označuje polohu a v rychlost nějaké částice, g vyjadřuje zrychlení působící na tuto částici. Budeme opět značit $r^{(k)} \approx r(t^{(k)})$ a $v^{(k)} \approx v(t^{(k)})$.

Princip metody leapfrog spočívá v tom, že pro výpočet hodnoty řešení v následujícím kroku je výhodnější použít jako odhad průměru derivace na intervalu $\langle t^{(k)}, t^{(k+1)} \rangle$ hodnotu derivace v jeho středu, než v jeho koncích. Předpis pro hodnotu polohy v následujícím kroku tedy zapíšeme jako

$$r^{(k+1)} = r^{(k)} + \Delta t \cdot v^{(k+\frac{1}{2})}, \tag{3.20}$$

kde $v^{(k+\frac{1}{2})}$ značí odhad rychlosti (tedy derivace polohy) v polovině intervalu.

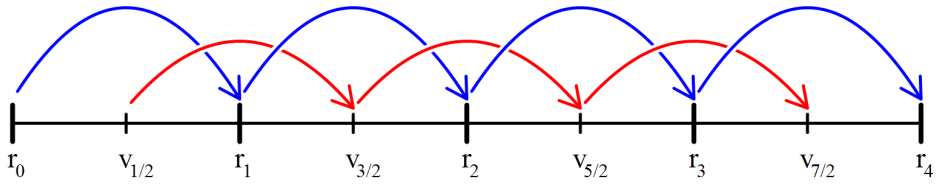
Hodnoty $v^{(k+\frac{1}{2})}$ ale také potřebujeme spočítat. Toho docílíme obdobným způsobem. Průměrnou derivaci funkce v na intervalu $\langle t^{(k+\frac{1}{2})}, t^{(k+\frac{3}{2})} \rangle$ odhadneme hodnotou derivace v jeho středu, tedy v bodě $t^{(k+1)}$. Při použití $v'(t^{(k)}) = g(r^{(k)})$ nabude předpis pro následující hodnotu rychlosti tvaru

$$v^{(k+\frac{3}{2})} = v^{(k+\frac{1}{2})} + \Delta t \cdot g(r^{(k+1)}). \tag{3.21}$$

Protože zpočátku známe pouze rychlost na začátku prvního intervalu, musíme rychlost v jeho polovině nějak dopočítat. K tomu použijeme nějakou jinou metodu, obvykle klasickou Eulerovu

$$v^{(\frac{1}{2})} = v^{(0)} + \frac{\Delta t}{2} g(r^{(0)}). \tag{3.22}$$

Pomocí předpisů (3.20) – (3.22) jsme schopni spočítat pozici částice ve všech potřebných krocích, nedovíme se ale její rychlost. Tu známe jen ve středech intervalů. Výpočet rychlosti



Obrázek 3.5: Metoda leapfrog

tedy rozdělíme na dvě části, přičemž v každé uděláme půl kroku. Výsledný předpis pro hodnotu polohy a rychlosti v následujícím kroku bude tedy mít tvar

$$\begin{aligned} v^{(k+\frac{1}{2})} &= v^{(k)} + \frac{\Delta t}{2} g(r^{(k)}), \\ r^{(k+1)} &= r^{(k)} + \Delta t \cdot v^{(k+\frac{1}{2})}, \\ v^{(k+1)} &= v^{(k+\frac{1}{2})} + \frac{\Delta t}{2} g(r^{(k+1)}). \end{aligned} \tag{3.23}$$

Může se zdát, že jsme tím zdvojnásobili počet vyhodnocení funkce g, ale není tomu tak. Hodnota funkce g z třetího řádku se totiž shoduje s hodnotou z prvního řádku v dalším kroku, můžeme ji tedy uložit a znovu použít. Právě popsanou upravenou metodu nazýváme rychlostní Verletova metoda.

Jestliže nás rychlost částice nezajímá, můžeme použít Verletův algoritmus, kterým získáme pouze pozice částice v jednotlivých krocích. Metoda je však dvoukroková, takže první krok musíme provést jinou metodou. Předpis pro následující pozici je

$$r^{(k+1)} = 2r^{(k)} - r^{(k+1)} + \Delta t^2 \cdot g(r^{(k)}). \tag{3.24}$$

Všechny zmíněné metody založené na bázi leapfrog jsou druhého řádu [21], což je v kombinaci s tím, že v jednom kroku provedeme pouze jedno vyhodnocení funkce g, výhodou. Dalšími výhodami těchto metod je, že zachovávají moment hybnosti a jsou časově reversibilní (tj. dostaneme stejné řešení, když algoritmus provedeme od konce na začátek). Nevýhodou je to, že nemůžeme pracovat s variabilním časový krokem.

Metoda leapfrog dostala svůj název zřejmě od toho, že střídavě počítáme pozice a rychlosti v navazujících krocích a mezikrocích, jak je vyobrazeno na obrázku 3.5. Vypadá to, jako by se navzájem přeskakovaly dvě žáby, odtud leapfrog.

3.4 Aplikace numerických metod na problém mnoha těles

Vraťme se k problému mnoha těles a řešení úlohy (2.8), (2.9) a popsané numerické metody aplikujme. Jak bylo zmíněno, čas, po který simulaci provádíme, rozdělíme na několik intervalů s ekvidistantními krajními body $\{t^{(k)}\}_{k=0}^{m}$. V těchto časových bodech postupně zjistíme polohy a rychlosti všech částic. Pracujeme ale s několika částicemi najednou, v předpisech jednotlivých metod tedy některé hodnoty nahradíme vektory a obdobně funkce nahradíme vektorovými funkcemi.

Začneme metodou leapfrog, u níž použijeme rychlostní Verletovu metodu. V rovnici (3.23) dosadíme za funkci g funkci $\boldsymbol{f}$, která představuje zrychlení působící na částice, a hodnoty r a v nahradíme příslušnými vektory pozic a rychlostí. Pro metodu leapfrog tedy dostaneme předpis

$$\begin{aligned} \boldsymbol{v}^{(k+\frac{1}{2})} &= \boldsymbol{v}^{(k)} + \frac{\Delta t}{2}\boldsymbol{f}(\boldsymbol{r}^{(k)}), \\ \boldsymbol{r}^{(k+1)} &= \boldsymbol{r}^{(k)} + \Delta t \cdot \boldsymbol{v}^{(k+\frac{1}{2})}, \\ \boldsymbol{v}^{(k+1)} &= \boldsymbol{v}^{(k+\frac{1}{2})} + \frac{\Delta t}{2}\boldsymbol{f}(\boldsymbol{r}^{(k+1)}). \end{aligned} \tag{3.25}$$

U ostatních metod transformujeme soustavu dvou diferenciálních rovnic (2.8) na pouze jednu diferenciální rovnici. Definujeme

$$\begin{aligned} \boldsymbol{y}(t) &= \begin{pmatrix} \boldsymbol{r}(t) \\ \boldsymbol{v}(t) \end{pmatrix}, \qquad & \boldsymbol{y}^{(k)} &= \begin{pmatrix} \boldsymbol{r}^{(k)} \\ \boldsymbol{v}^{(k)} \end{pmatrix}, \\ \boldsymbol{g}(t, \boldsymbol{y}(t)) &= \boldsymbol{g}\left(t, \begin{pmatrix} \boldsymbol{r}(t) \\ \boldsymbol{v}(t) \end{pmatrix}\right) = \begin{pmatrix} \boldsymbol{v}(t) \\ \boldsymbol{f}(\boldsymbol{r}(t)) \end{pmatrix}. \end{aligned} \tag{3.26}$$

Soustavu (2.8) pak můžeme zapsat ve tvaru

$$\boldsymbol{y}'(t) = \boldsymbol{g}(t, \boldsymbol{y}(t)). \tag{3.27}$$

Při těchto definicích je použití Eulerovy metody a metody Rungeho-Kutta vcelku přímočaré. V předpisech dosadíme za hodnoty $y^{(k)}$ a funkci g jejich vektorové varianty definované výše a počítáme vektorově.

Pro Eulerovu metodu použijeme její dopřednou variantu. Do předpisu (3.7) dosadíme vztahy pro $y^{(k)}$ a g a upravíme,

$$\begin{pmatrix} \boldsymbol{r}^{(k+1)} \\ \boldsymbol{v}^{(k+1)} \end{pmatrix} = \boldsymbol{y}^{(k+1)} = \boldsymbol{y}^{(k)} + \Delta t \cdot \boldsymbol{g}(t^{(k)}, \boldsymbol{y}^{(k)}) = \begin{pmatrix} \boldsymbol{r}^{(k)} \\ \boldsymbol{v}^{(k)} \end{pmatrix} + \Delta t \cdot \begin{pmatrix} \boldsymbol{v}^{(k)} \\ \boldsymbol{f}(\boldsymbol{r}^{(k)}) \end{pmatrix}. \tag{3.28}$$

Předpis pro následující hodnoty pozic a rychlostí částic podle Eulerovy metody má tedy tvar

$$\begin{aligned} \boldsymbol{r}^{(k+1)} &= \boldsymbol{r}^{(k)} + \Delta t \cdot \boldsymbol{v}^{(k)}, \\ \boldsymbol{v}^{(k+1)} &= \boldsymbol{v}^{(k)} + \Delta t \cdot \boldsymbol{f}(\boldsymbol{r}^{(k)}). \end{aligned} \tag{3.29}$$

Z Rungeho-Kuttových metod použijeme metodu čtvrtého řádu. Koeficienty k_1 – k_4 se stanou vektory a získají tvar

$$\begin{aligned} \begin{pmatrix} \boldsymbol{k}_{11} \\ \boldsymbol{k}_{12} \end{pmatrix} &= \boldsymbol{k}_1 = \boldsymbol{g}(t^{(k)}, \boldsymbol{y}^{(k)}) = \cdots = \begin{pmatrix} \boldsymbol{v}^{(k)} \\ \boldsymbol{f}(\boldsymbol{r}^{(k)}) \end{pmatrix}, \\ \begin{pmatrix} \boldsymbol{k}_{21} \\ \boldsymbol{k}_{22} \end{pmatrix} &= \boldsymbol{k}_2 = \boldsymbol{g}(t^{(k)} + \frac{\Delta t}{2}, \boldsymbol{y}^{(k)} + \frac{\Delta t}{2}\boldsymbol{k}_1) = \cdots = \begin{pmatrix} \boldsymbol{v}^{(k)} + \frac{\Delta t}{2} \cdot \boldsymbol{k}_{12} \\ \boldsymbol{f}(\boldsymbol{r}^{(k)} + \frac{\Delta t}{2} \cdot \boldsymbol{k}_{11}) \end{pmatrix}, \\ \begin{pmatrix} \boldsymbol{k}_{31} \\ \boldsymbol{k}_{32} \end{pmatrix} &= \boldsymbol{k}_3 = \boldsymbol{g}(t^{(k)} + \frac{\Delta t}{2}, \boldsymbol{y}^{(k)} + \frac{\Delta t}{2}\boldsymbol{k}_2) = \cdots = \begin{pmatrix} \boldsymbol{v}^{(k)} + \frac{\Delta t}{2} \cdot \boldsymbol{k}_{22} \\ \boldsymbol{f}(\boldsymbol{r}^{(k)} + \frac{\Delta t}{2} \cdot \boldsymbol{k}_{21}) \end{pmatrix}, \\ \begin{pmatrix} \boldsymbol{k}_{41} \\ \boldsymbol{k}_{42} \end{pmatrix} &= \boldsymbol{k}_4 = \boldsymbol{g}(t^{(k)} + \Delta t, \boldsymbol{y}^{(k)} + \Delta t\boldsymbol{k}_3) = \cdots = \begin{pmatrix} \boldsymbol{v}^{(k)} + \Delta t \cdot \boldsymbol{k}_{32} \\ \boldsymbol{f}(\boldsymbol{r}^{(k)} + \Delta t \cdot \boldsymbol{k}_{31}) \end{pmatrix}. \end{aligned} \tag{3.30}$$

Předpisy pro následující hodnoty pozic a rychlostí částic tedy jsou

$$\begin{aligned} \boldsymbol{r}^{(k+1)} &= \boldsymbol{r}^{(k)} + \Delta t \cdot \frac{\boldsymbol{k}_{11} + 2\boldsymbol{k}_{21} + 2\boldsymbol{k}_{31} + \boldsymbol{k}_{41}}{6}, \\ \boldsymbol{v}^{(k+1)} &= \boldsymbol{v}^{(k)} + \Delta t \cdot \frac{\boldsymbol{k}_{12} + 2\boldsymbol{k}_{22} + 2\boldsymbol{k}_{32} + \boldsymbol{k}_{42}}{6}. \end{aligned} \tag{3.31}$$

Tím, jak efektivně vyhodnotit funkci $\boldsymbol{f}$ (tedy zrychlení působící na částice), se budeme zabývat v kapitole 4.

4 Algoritmy pro výpočet sil

V této kapitole se budeme zabývat tím, jak efektivně vypočítat síly[2] působící na částice v jednotlivých krocích simulace. Pro výpočet sil existuje řada algoritmů, které se liší časovou složitostí i přesností [2, 13, 19]. Nejjednodušší, nejpřesnější, ale zato nejnáročnější na výpočetní čas jsou přímé metody (particle-particle) se složitostí $\mathcal{O}(n^2)$. Metody využívající stromovou strukturu (tree code) jsou schopny malým snížením přesnosti zlepšit časovou složitost na $\mathcal{O}(n \log n)$. Metody, které částice diskretizují na mřížkovou strukturu (particle-mesh), mají složitost $\mathcal{O}(n)$ vůči počtu částic a $\mathcal{O}(G \log G)$ vůči počtu bodů mřížky. Vrcholem efektivity jsou metody vylepšující stromovou metodu (fast multipole) se složitostí $\mathcal{O}(n)$. V této práci se budeme detailněji zabývat přímou metodou a zejména Barnesovým-Hutovým algoritmem, příkladem metody, která používá stromovou strukturu.

4.1 Zjemňování síly, softening

Síly působící na částice bychom mohli počítat podle rovnice (2.1). Tato rovnice však přináší problém se singularitou, nacházejí-li se dvě částice velmi blízko sebe, tedy když $\boldsymbol{r}_i \to \boldsymbol{r}_j$ a tudíž $\|\boldsymbol{F}_{ij}\| \to \infty$. Protože by byla síla příliš velká, částicím by bylo dáno příliš velké zrychlení.

V reálném světě by to nebyl problém, protože by obdrženou rychlost následně ztratily tím, že by se dostaly do opačné konfigurace, čímž by dostaly zrychlení stejné velikosti, ale opačného směru. V iteračních simulacích ale k takovému procesu (většinou) nedojde. Částice se nadměrně zrychlí, jejich rychlost bude vůči sobě opačná a velmi velká, tudíž se v další iteraci budou nacházet příliš daleko od sebe na to, aby vzájemnou interakcí rychlost zase snížily. Nadměrná rychlost jim tedy zůstane. Částice pak zpravidla opustí simulovaný systém. Dochází k velké chybě a nereálnému chování systému.

Pro vyřešení tohoto problému se využívá zjemňování síly – softening [19]. Použijeme Plummerovo zjemňování (Plummer softening). Modifikujeme standardní vzorec pro výpočet vzdálenosti dvou částic tak, aby nikdy nenabýval nulové hodnoty

$$\rho_{ij} = \sqrt{\|\boldsymbol{r}_{ij}\|^2 + \epsilon^2}, \tag{4.1}$$

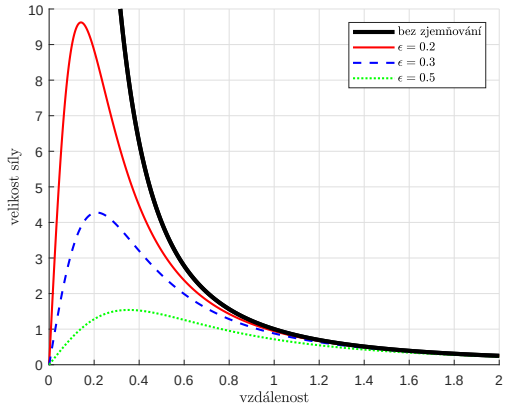
kde $\epsilon > 0$ je zjemňovací parametr. Rovnice pro výpočet síly mezi dvěma částicemi pak získá tvar

$$\boldsymbol{F}_{ij} = G\frac{m_i m_j}{\rho_{ij}^3}\boldsymbol{r}_{ij} = G\frac{m_i m_j}{(\|\boldsymbol{r}_{ij}\|^2 + \epsilon^2)^{3/2}}\boldsymbol{r}_{ij}. \tag{4.2}$$

Graf síly v závislosti na vzdálenosti částic pro různé volby parametru ϵ je vykreslen na obrázku 4.1. Je zde také vidět, že pro vzdálenost mnohem větší než nula je zjemněná síla téměř shodná s nezjemněnou.

Díky tomu, že modifikovaná vzdálenost nikdy nebude nulová (a síla tudíž zůstane konečná), jsme se zbavili singularity a s tím spojených chyb. Vytvořili jsme tím ale další chybu – na blízké vzdálenosti je hodnota vypočtené síly nižší, čímž se potlačí blízké interakce mezi částicemi – kolize. Takovým simulacím, které sledují zejména vývoj systému jako celku a blízké interakce nejsou důležité, se říká bezkolizní (collisionless). Pokud jsou podstatné i blízké interakce, mluvíme o kolizním systému (collisional) [4]. Optimální hodnota zjemňovacího parametru ϵ je pro

[2] potřebujeme znát zrychlení, v této kapitole se však budeme zabývat výpočtem sil



Obrázek 4.1: Porovnání silových funkcí pro různé hodnoty zjemňovacího parametru ϵ

bezkolizní simulace rovna přibližně polovině střední vzdálenosti částic v nejhustších částech systému [17].

4.2 Přímá metoda particle-particle

Jak název napovídá, metoda particle-particle počítá síly mezi všemi dvojicemi částic v systému. Jedná se tedy o prostou evaluaci pravé strany rovnice (4.2) pro každou dvojici i a j, $i \neq j$. Můžeme si ale všimnout, že $\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji}$, díky čemuž můžeme snížit počet potřebných výpočtů síly z $n \cdot (n-1)$ na polovinu. Časová složitost však zůstane v řádu $\mathcal{O}(n^2)$. Princip metody particle-particle je ukázán v algoritmu 1.

Algoritmus 1 Metoda particle-particle

```
function CalculateForcesPP(particles, n)
    for i from 0 to n-1 do
        forces[i] ← 0⃗
        for j from 0 to n-1 do
            if i ≠ j then
                forces[i] ← forces[i] + CalculateForce(particles[i], particles[j])
            end if
        end for
    end for
end function
```

4.3 Barnesův-Hutův algoritmus

Barnesův-Hutův algoritmus [1] aproximuje shluky částic, které jsou dostatečně malé a od uvažované částice dostatečně vzdálené, jedinou virtuální částicí. Tím za cenu snížení přesnosti dosahuje lepší časové složitosti $\mathcal{O}(n \log n)$. Celý proces výpočtu sil lze rozdělit do tří částí:

1. Konstrukce stromu, rozdělení částic do krychlových buněk
2. Výpočet vlastností buněk
3. Samotný výpočet sil

Všechny tyto části se pro každý výpočet sil provádějí znovu. Nebudeme algoritmus komplikovat tím, že bychom nechali strom sestrojený a částice bychom v jednotlivých krocích složitě přemisťovali.

Částice v systému uspořádáme do stromové struktury – adaptivního octree. Uzly ve stromu reprezentují krychle v simulovaném prostoru, kterým říkáme buňky. Každá buňka má nejvýše osm potomků – podbuněk, menších krychlí, které se uvnitř ní nacházejí. Vnitřní buňky představují oktanty větší buňky a mají oproti ní poloviční velikost hrany.

V každé buňce se nacházejí všechny částice, které náleží do příslušné části prostoru, a v každé koncové, dále nedělené buňce, se nachází maximálně jedna částice. Každá buňka má kromě svých potomků také několik vlastností – souřadnice geometrického středu, velikost hrany, celkovou hmotnost obsažených částic a souřadnice hmotného středu.

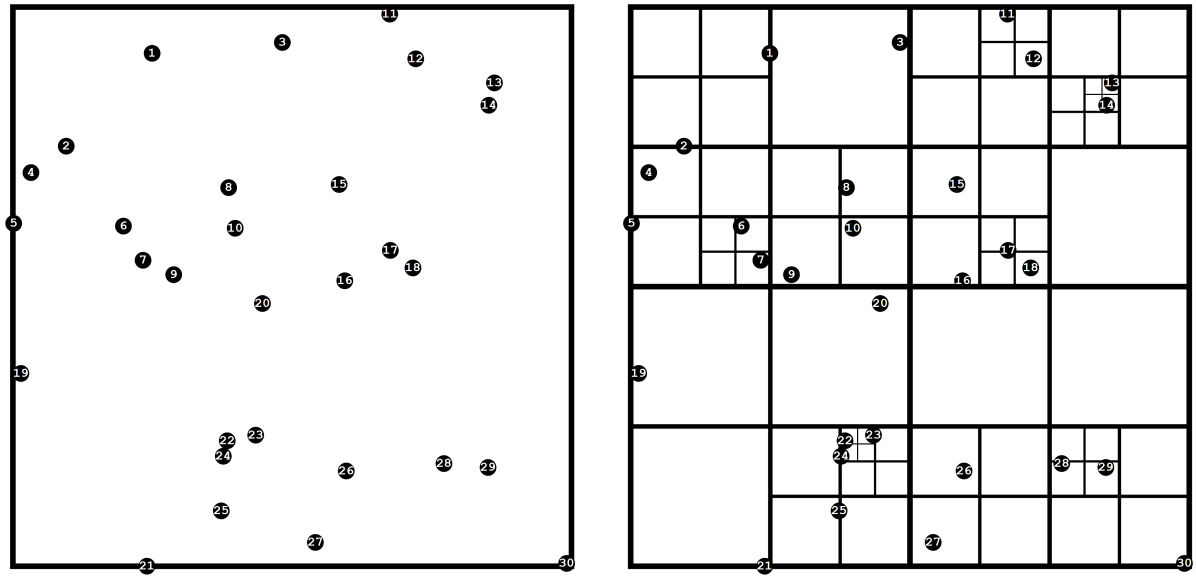
4.3.1 Konstrukce stromu

Konstrukci stromu je možné provést více způsoby. První možnost je začít s jednou prázdnou buňkou, dostatečně velkou na to, aby byla schopna pojmout všechny částice. Poté postupně všechny částice po jedné vkládáme do této první, kořenové buňky. Pokud částici vložíme do buňky, která je již rozdělena na podbuňky, vložíme ji do příslušné podbuňky a rekurzivně opakujeme. Pokud částici vložíme do koncové (dále nedělené) buňky, kde se již nějaká částice nachází, tuto buňku rozdělíme na osm podbuněk a obě částice vložíme do příslušných podbuněk. Pokud částici vložíme do prázdné buňky, proces vkládání této částice končí. Po vložení všech částic můžeme všechny prázdné buňky odstranit.

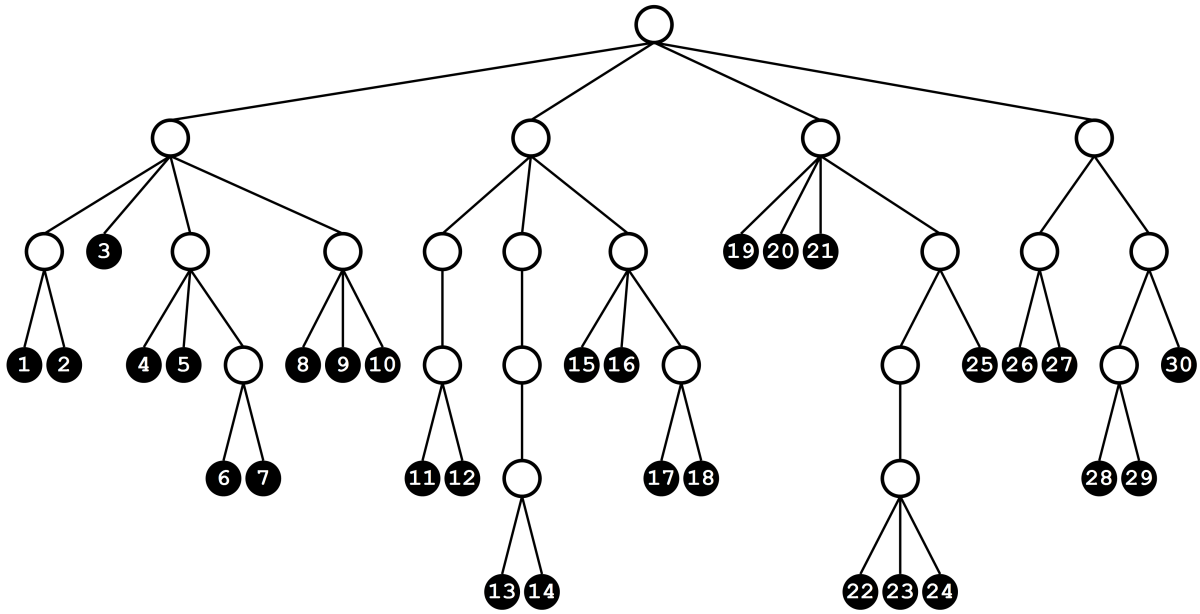
Druhou možností je opět začít s dostatečně velkou kořenovou buňkou schopnou pojmout všechny částice. Všechny částice, které se v oblasti buňky nacházejí, roztřídíme do osmi kategorií podle toho, do kterého oktantu dané buňky náleží. Pro každou neprázdnou kategorii pak vytvoříme novou podbuňku, do které přiřadíme všechny příslušné částice, a postup rekurzivně opakujeme, dokud se v buňce nachází více než jedna částice.

Výška vytvořeného stromu (pokud není zdegenerovaný[3]) je v řádu $\mathcal{O}(\log n)$. Při vložení jedné částice projdeme maximálně všemi úrovněmi stromu, jeho konstrukce má tedy časovou složitost $\mathcal{O}(n \log n)$. Počet uzlů ve stromu je řádově $\mathcal{O}(n)$. Na obrázku 4.2 je (pro jednoduchost ve dvou dimenzích) znázorněno rozdělení prostoru s částicemi do čtvercových buněk. Struktura příslušného stromu (adaptivní quadtree) je pak vyobrazena na obrázku 4.3, kde prázdný kruh představuje dále dělenou buňku – vnitřní uzel – a vyplněný kruh reprezentuje koncový uzel s částicí. Indexy částic mezi obrázky korespondují.

[3] tzn. málo rozvětvený strom připomínající spíše cestu



Obrázek 4.2: Rozdělení systému částic do čtvercových buněk



Obrázek 4.3: Vytvořený quadtree

4.3.2 Výpočet vlastností buněk

Informace o poloze geometrického středu buňky a její velikosti se ukládají již při jejím vytváření. Naopak výpočet celkové hmotnosti a polohy hmotného středu buňky provedeme až když bude celý strom vytvořený.

Začneme od koncových buněk, které obsahují pouze jednu částici. U nich je poloha hmotného středu a hmotnost buňky zřejmá – je to poloha a hmotnost částice, kterou obsahují.

Poté postupujeme od koncových buněk nahoru až ke kořenové buňce. Abychom mohli spočítat vlastnosti nějaké buňky, musíme již mít vypočtené vlastnosti všech jejích podbuněk. Celkovou hmotnost buňky spočítáme jako součet hmotností všech jejích podbuněk

$$M = \sum_{i=1}^{8} M_i, \tag{4.3}$$

kde M je celková hmotnost buňky a M_i je hmotnost i-té podbuňky. Pokud je podbuňka prázdná, její hmotnost je nulová. Hmotný střed buňky počítáme jako vážený průměr hmotných středů všech podbuněk, kde váhy jsou hmotnosti příslušných podbuněk

$$\boldsymbol{R} = \frac{1}{M} \cdot \sum_{i=1}^{8} M_i \boldsymbol{R}_i, \tag{4.4}$$

kde $\boldsymbol{R}$ je hmotný střed celé buňky a $\boldsymbol{R}_i$ je hmotný střed i-té podbuňky. Pokud je podbuňka prázdná, její hmotnost je nulová a na pozici jejího hmotného středu nezáleží.

Tento proces má časovou složitost pouze $\mathcal{O}(n)$ – strom obsahuje řádově $\mathcal{O}(n)$ uzlů, každý uzel stromu projdeme právě jednou, přičemž v něm provedeme konstantní počet operací.

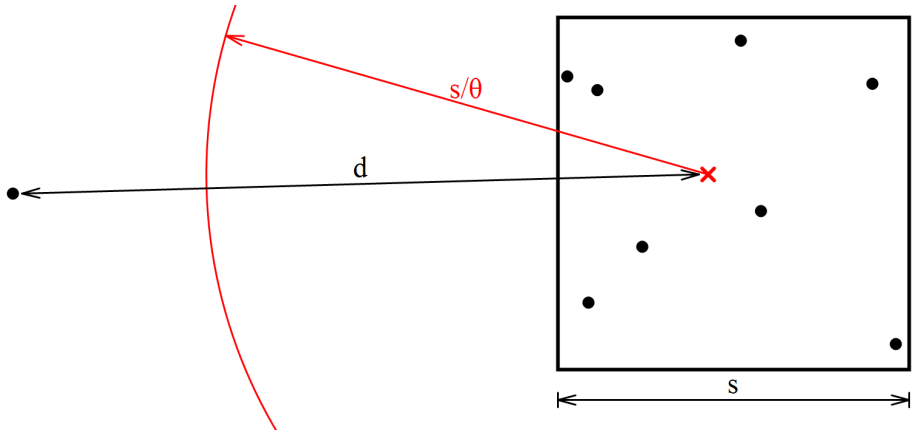
4.3.3 Výpočet sil

Mějme zadaný parametr přesnosti θ, jehož hodnota je obvykle volena v rozmezí $0{,}5 \leq \theta \leq 1$ [14, 16]. Pro tento parametr platí, že čím je jeho hodnota vyšší, tím větší chyby se dopustíme, ale na druhou stranu výpočet sil trvá kratší dobu. Naopak čím je θ menší, tím přesnější a časově náročnější výpočet je.

Nyní máme vše připraveno pro to, abychom mohli začít s výpočtem samotných sil. Při výpočtu síly působící na nějakou částici rekurzivně projdeme vytvořený strom, přičemž začneme od největší, kořenové buňky a postupujeme hlouběji do stromu.

Uvažujme, že právě porovnáváme částici s nějakou buňkou, která má velikost hrany s a vzdálenost jejího hmotného středu od dané částice je d. Pokud platí podmínka $s/d < \theta$, tedy buňka je dostatečně malá a dostatečně vzdálená, všechny částice v buňce aproximujeme jedinou virtuální částicí s pozicí v hmotném středu buňky a hmotností rovnou hmotnosti buňky a vůči této virtuální částici spočítáme sílu. Pokud podmínka neplatí, pak tento proces rekurzivně provedeme pro všechny podbuňky této buňky. Síly, kterými na částici jednotlivé podbuňky působí, poté sečteme. Pokud se v buňce nachází pouze jedna částice, provedeme výpočet síly vůči ní. Síla, kterou na částici působí prázdná buňka, je patrně nulová. Pseudokód k výpočtu síly, kterou působí buňka na jednu částici, je ukázán v algoritmu 2.

Na obrázku 4.4 jsou vpravo zobrazeny částice v buňce, jejíž hmotný střed je vyznačen červeným křížkem, a vlevo je částice, která s buňkou interaguje. Červený oblouk představuje hranici



Obrázek 4.4: Částice interagující s buňkou

aproximace – všechny částice vně oblasti buňku aproximují jako jednu virtuální částici. Na obrázku je voleno $\theta = 0{,}7$.

Díky tomu, že nepočítáme síly mezi každou dvojicí částic, ale některé skupiny částic pro tento účel aproximujeme jednou virtuální částicí, čímž snížíme počet potřebných interakcí, má výpočet sil časovou složitost pouze $\mathcal{O}(n \log n)$ [1]. Při volbě $\theta = 0$ však algoritmus výpočtu sil zdegeneruje v algoritmus particle-particle a složitost se zhorší na $\mathcal{O}(n^2)$.

Algoritmus 2 Metoda výpočtu síly podle Barnese a Huta

```
function CalculateForceBHOneParticle(particle, currCell, θ)
    if IsLeafCell(currCell) then
        force ← CalculateForce(particle, currCell.particle)
    else
        distance ← DistanceBetween(particle.position, currCell.centerOfMass)
        if currCell.size / distance < θ then
            force ← CalculateForce(particle, currCell.virtualParticle)
        else
            force ← 0⃗
            for all subCells in currCell do
                force ← force + CalculateForceBHOneParticle(particle, subCell, θ)
            end for
        end if
    end if
    return force
end function
```

5 Implementace

V této kapitole je probrána implementace popsaných algoritmů a dalšího nezbytného kódu. Program je napsán v jazyce C++, který je spolu s Fortranem nejčastěji používaným jazykem pro náročné numerické simulace. Zdrojový kód byl psán ve Visual Studiu 2017 (ve kterém také probíhalo základní testování). Pro účely robustnějšího testování je kód přeložitelný i na linuxových systémech.

Nejdůležitější částí je projekt `ParticleSimulation` řešící samotné provádění simulace. Pro generování počátečních podmínek a jejich úpravu jsou vytvořeny programy `UniverseGenerator` a `UniverseModifier`. `Converter` má na starosti konverzi souborů mezi různými formáty, mimo jiné i do formátu `*.vtk` [11] vhodného pro vizualizaci simulací v programu Paraview. Program `ErrorAnalyzer` vyhodnocuje chyby a nepřesnosti simulace. Pokud není upřesněno jinak, věnujeme se v podkapitolách implementaci `ParticleSimulation`. Veškeré zdrojové kódy jsou přiloženy k této práci.

5.1 Struktura programu

Hlavní třídou, která řídí celou simulaci, je třída `Simulation`. Ta obsahuje jednu instanci třídy `ParticleCollectionSequence`, ve které jsou uloženy hmotnosti všech částic a jejich stavy (polohy a rychlosti) v jednotlivých krocích. Stav částic v jednom kroku simulace udržuje třída `ParticleCollection`, která obsahuje dvě instance třídy `VectorCollection` – jednu pro polohy částic a druhou pro jejich rychlosti. Na obrázku 5.1 je diagram zobrazující třídy v programu a vztahy mezi nimi.

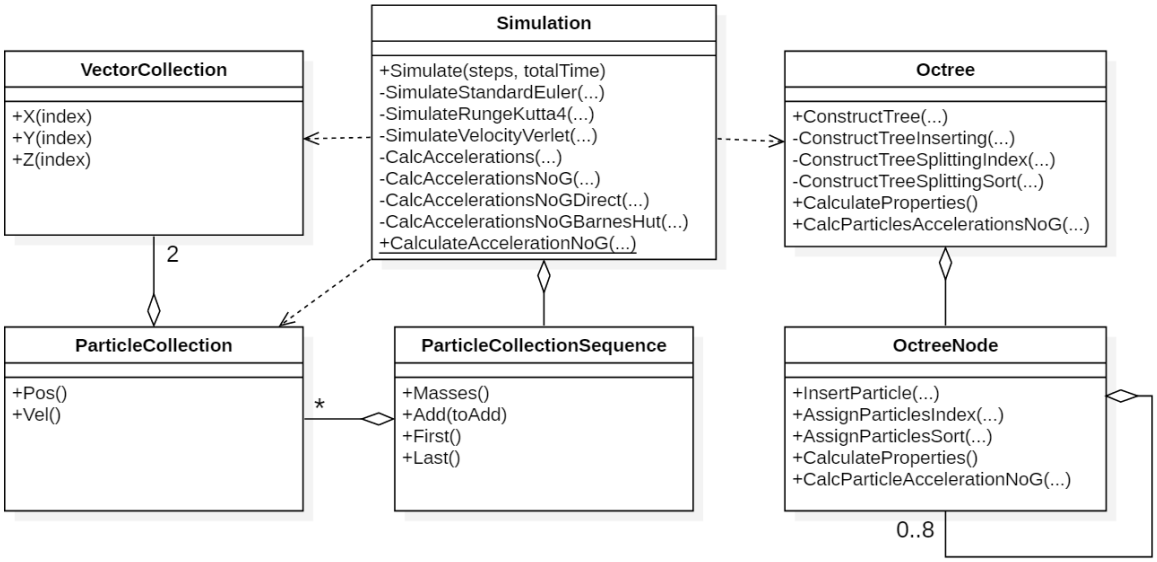
Informace o tom, jak bude simulace probíhat, tedy jaké metody a algoritmy se použijí, jsou uloženy ve statické třídě `SimulationConfig`. Lze nastavit metodu numerického řešení počáteční úlohy (klasická Eulerova metoda, metoda Rungeho-Kutta čtvrtého řádu, rychlostní Verletova metoda), algoritmus výpočtu zrychlení (particle-particle, Barnesův-Hutův algoritmus) a jeho variantu (pro particle-particle vyhodnocení všech n^2 zrychlení nebo jen poloviny, pro Barnesův-Hutův algoritmus způsob konstrukce stromu a parametr θ), zjemňovací parametr ϵ a další méně podstatné parametry. Vše lze nastavit při spuštění programu pomocí argumentů příkazové řádky. Tím se podrobněji zabýváme v podkapitole 5.6.

Metoda, která samotnou simulaci provádí, je pojmenována `Simulate`. Její parametry jsou počet časových kroků a celkový simulační čas. Tato metoda podle nastavení simulace rozhodne, který numerický řešič se použije, a zavolá příslušnou metodu, která simulaci provede. Všechny numerické řešiče jednotně volají metodu `CalcAccelerations`, která zvoleným algoritmem vypočte zrychlení. Implementací algoritmů pro výpočet zrychlení se budeme zabývat v podkapitole 5.4.

V programu je používán datový typ `FloatType`, což je alias za typ `double`, nebo `float`. Tento alias je spolu s dalšími globálně platnými definicemi a makry definován v hlavičkovém souboru `Definitions.h`.

5.2 Pole struktur, struktura polí

Třída `VectorCollection` v programu reprezentuje kolekci třísložkových vektorů. Tuto třídu je možné implementovat více způsoby [10]. Intuitivní možností je mít jedno pole, jehož prvky jsou jednotlivé vektory (obecně nějaké struktury). Takový přístup nazýváme pole struktur nebo



Obrázek 5.1: Třídy v programu

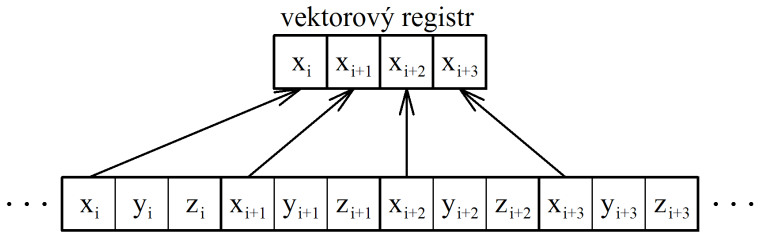
Array of Structures (AoS). Druhou možnosti je mít zvlášť pole pro každou složku vektoru. Této možnosti říkáme struktura polí nebo Structure of Arrays (SoA). Existují i další způsoby, budeme se ale zabývat jen těmito dvěma.

Použití SoA je často efektivnější, a to kvůli způsobu, jakým jsou data načítána do vektorové jednotky moderních procesorů. Ta dokáže provést stejnou operaci nad několika páry operandů současně, což je výhodnější, než provedení operací jednotlivě. Pokud používáme SoA, nacházejí se stejné složky sousedních vektorů přímo vedle sebe, kdežto u AoS jsou mezi nimi další hodnoty. Načítání dat do vektorové jednotky lze tedy při použití SoA provést efektivněji a rychleji. Výkonnostní rozdíl mezi AoS a SoA experimentálně ověříme v kapitole 6.4. Na obrázcích 5.2 a 5.3 je zobrazeno schéma struktury paměti a způsob načítání dat do registru vektorové jednotky.

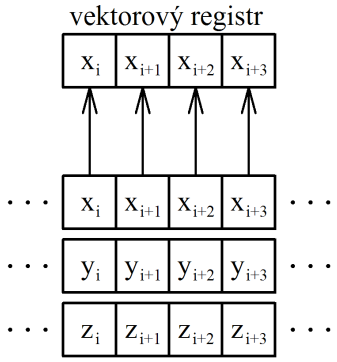
V kódu ale obvykle přímo neprovádíme stejné operace s několika proměnnými, které jsou v paměti uspořádány za sebou. Kompilátor však dokáže v některých případech kód transformovat tak, aby byl program schopný vektorovou jednotku využít. To se děje nejvíce u cyklů, kdy sekvenčně zpracováváme celé pole dat.

Třída `VectorCollection` je implementována oběma zmíněnými způsoby. K rozhodnutí, která varianta se použije, dochází při překladu programu podle hodnoty makra `VECTORS_IN_MEMORY`. Třída má pro přístup k jednotlivým hodnotám metody s jednotným předpisem, takže není nutné psát veškerý kód pro obě možnosti zvlášť. Tyto metody by měl překladač inlinovat[4], takže by nemělo docházet ke ztrátě na výkonu. Podstatná část třídy `VectorCollection` je zobrazena ve výpisu kódu 5.1.

[4]tzn. volání funkce nahradit jejím tělem



Obrázek 5.2: Pole struktur, array of structures, AoS



Obrázek 5.3: Struktura polí, structure of arrays, SoA

```
1  class VectorCollection
2  {
3  private:
4      int count;
5  #if VECTORS_IN_MEMORY == ARRAY_OF_STRUCTURES
6      Vector* vectors;
7  #elif VECTORS_IN_MEMORY == STRUCTURE_OF_ARRAYS
8      FloatType* x;
9      FloatType* y;
10     FloatType* z;
11 #endif
12 public:
13     // ...
14     FloatType& X(int index);
15     // ...
16 };
```

Výpis kódu 5.1: Data v třídě VectorCollection

5.3 Paralelizace

Abychom využili plný potenciál dnešních počítačových systémů, je třeba program paralelizovat. Využitím všech dostupných procesorových jader snížíme dobu vykonávání programu. Paralelní systémy lze rozdělit podle vícero kritérií [3].

Základní dělení paralelismu poskytuje Flynnova taxonomie. Ta dělí architektury podle počtu paralelně zpracovávaných instrukcí a dat. Nejjednodušší je SISD (Single Instruction Single Data) reprezentující klasickou neparalelní architekturu. SIMD (Single Instruction Multiple Data) zařízení mají jen jeden proud instrukcí, jedna instrukce však může provést operaci nad větším množstvím dat současně. Příkladem je procesor s dříve zmíněnou vektorovou jednotkou nebo grafická karta. MIMD (Multiple Instruction Multiple Data) je schopná současně obsluhovat více instrukčních proudů (vláken) pracujících s různými daty na více procesorových jádrech.

MIMD lze dále rozdělit na systémy se sdílenou pamětí a systémy s distribuovanou pamětí. Systém s distribuovanou pamětí si lze představit jako několik počítačů (uzlů), které jsou vzájemně propojené komunikační sítí. Každý uzel je systémem se sdílenou pamětí a má přímý přístup jen do své lokální paměti. Komunikace mezi procesy probíhá explicitně přes posílání zpráv, pro které existuje standard MPI (Message Passing Interface). Komunikace bývá u těchto systémů úzkým hrdlem.

Systém se sdílenou pamětí lze reprezentovat například počítačem s jedním nebo dvěma vícejádrovými procesory. Všechna procesorová jádra mají přímý přístup do celé operační paměti, podmínky přístupu (latence, propustnost[5]) do různých částí paměti však nemusí být pro všechna jádra stejné. Pokud stejné jsou, jedná se o UMA (Uniform Memory Access) systém. V opačném případě mluvíme o NUMA (NonUniform Memory Access) systému. Nejpoužívanějším standardem pro paralelizaci programů pro tyto systémy je OpenMP.

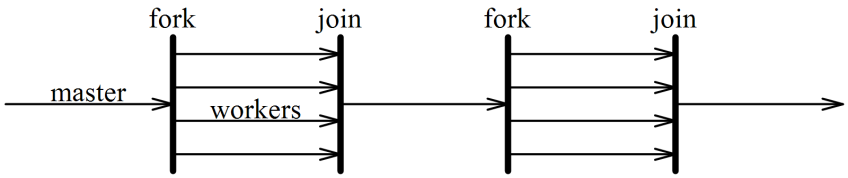
Principy distribuovaného systému lze implementovat na systému se sdílenou pamětí a naopak, nemusí to však být vždy výhodné. Systém nemusí výlučně patřit jen do jedné kategorie, můžeme mít program s více současně běžícími vlákny (MIMD), kde každé vlákno na svém jádře využívá vektorovou jednotku (SIMD).

5.3.1 OpenMP

Vytvořený program je paralelizován s použitím standardu OpenMP. Ten mimo jiné definuje jednoduché konstrukce pro paralelizaci cyklů a konstrukce, kterými je možné paralelizovat rekurzivní metody [3]. OpenMP funguje na principu fork-join modelu. Na začátku programu se spustí jedno hlavní vlákno (master thread). To pak může pro paralelní provedení nějakého kódu vytvořit a spustit (fork) několik pracovních vláken (worker threads). Až všechna vlákna dokončí přidělenou práci, hlavní vlákno pokračuje v sekvenčním vykonávání programu (join). Fork-join model je ilustrován na obrázku 5.4.

Základem celého OpenMP je direktiva překladače `#pragma omp parallel`. Takto označený blok kódu je spuštěn paralelně na více vláknech, přičemž každé vlákno vykonává stejný kód uvnitř bloku. V bloku je pak možné používat například funkce, které vrací počet spuštěných vláken a ID aktuálního vlákna. Buď můžeme pomocí těchto funkcí manuálně přiřadit každému vláknu jeho práci, nebo využijeme dalších OpenMP konstrukcí, které to vyřeší za nás.

[5] doba mezi žádostí o data a jejich načtením, resp. počet přenesených bajtů za vteřinu



Obrázek 5.4: Fork-join model

```
#pragma omp parallel for
    for (int i = 0; i < count; i++)
    {
        array[i] *= 2;
    }
```

Výpis kódu 5.2: Paralelizace cyklu for

```
#pragma omp parallel for schedule(static, 16)
    for (int i = 0; i < count; i++)
    {
        array[i] *= 2;
    }
```

Výpis kódu 5.3: Paralelizace cyklu for s použitím plánování

Paralelizace cyklů

Klasickým příkladem je paralelizace cyklu `for`. Ten musí být v kanonické formě (tj. klasický `for (initialize; test; increment) {...}` s několika restrikcemi), uvnitř cyklu se nesmějí nacházet příkazy, které by jej mohly předčasně ukončit a jeho iterace na sobě musí být nezávislé. Pro paralelizaci takového cyklu jej označíme `#pragma omp for` uvnitř paralelního bloku, nebo můžeme použít jedinou samostatnou konstrukci, viz výpis kódu 5.2. OpenMP samo cyklus rozdělí na několik částí (dávek, chunks), které jsou zpracovávány paralelně.

Plánování, scheduling

Způsob, jakým jsou dávky přiřazovány jednotlivým vláknům, a volitelně i velikost jedné dávky je možné ovlivnit pomocí klauzule `schedule` (odtud scheduling, plánování). Způsob přiřazování dávek vláknům může být buď `static`, `dynamic`, `guided`, `auto`, nebo `runtime`. Statické plánování rovnoměrně rozřadí dávky mezi jednotlivá vlákna cyklickým způsobem. Dynamické plánování probíhá až za běhu – kdykoli některé vlákno dokončí svou práci, je mu přidělena další dosud nezpracovaná dávka. Guided je podobné dynamickému, velikost dávky ale začíná na větší hodnotě a postupně se snižuje. Auto nechává rozhodnutí o výběru plánování na překladači. Runtime znamená, že se způsob plánování vybere až za běhu podle hodnoty vnitřních proměnných. Ukázka cyklu `for` s použitím statického plánování s velikostí dávky 16 je zobrazena ve výpisu kódu 5.3.

```
1      int sum = 0;
2  #pragma omp parallel for
3      for (int i = 0; i < count; i++)
4      {
5  #pragma omp atomic
6          sum += array[i];
7      }
```

Výpis kódu 5.4: Paralelní sčítání čísel s použitím atomic

```
1      int sum = 0;
2  #pragma omp parallel for reduction(+: sum)
3      for (int i = 0; i < count; i++)
4      {
5          sum += array[i];
6      }
```

Výpis kódu 5.5: Paralelní sčítání s použitím redukce

Critical, atomic

Uvažujme, že chceme paralelně sečíst celé pole čísel. Prostá paralelizace cyklu přičítajícího čísla do společné proměnné by byla chybná. Mohlo by totiž dojít k souběhu (race condition), tedy k situaci, kdy více vláken současně zapisuje do jedné proměnné, čímž vznikají chyby. Pro tyto případy existuje v OpenMP konstrukce `critical`. Takto označený blok kódu je prováděn vždy maximálně jedním vláknem současně. Ostatní vlákna, která chtějí do kritické sekce vstoupit, musejí čekat, než se kritická sekce uvolní. Použití `critical` ale přináší významnou režii (overhead) nutnou ke komunikaci mezi vlákny. V OpenMP existuje konstrukce `atomic`, viz výpis kódu 5.4. Ta má režii daleko menší a zajišťuje, že operace s pamětí v další instrukci bude provedena atomicky. Data se načtou, aktualizují a zapíší najednou. Nic nemůže do paměti přistupovat ani atomickou operaci přerušit, dokud není dokončena. Atomické operace bývají podporovány přímo hardwarem.

Redukce

Konstrukce `atomic` je však stále zatížena nějakou režií. Pro paralelní sčítání (a jiné operace) existuje v OpenMP konstrukce `reduction`, viz výpis kódu 5.5, která má režii zanedbatelnou. Je třeba specifikovat operaci a proměnné, na kterých se redukce provede. V cyklu se hodnoty přičítají do proměnné privátní pro dané vlákno, na konci cyklu je pak proveden jejich vzájemný součet. K souběhu tedy nedochází a režie je minimální.

Vektorizace, SIMD

Současné kompilátory jsou do určité míry schopny některé cykly automaticky vektorizovat. OpenMP konstrukce `#pragma omp simd` překladači napoví, že by následující cyklus měl být

```
1  int FibNum(int num)
2  {
3      int x, y;
4      if(num <= 1)
5          return num;
6  #pragma omp task shared(x)
7      x = FibNum(num - 1);
8  #pragma omp task shared(y)
9      y = FibNum(num - 2);
10 #pragma omp taskwait
11     return x + y;
12 }
```

Výpis kódu 5.6: Paralelizace výpočtu Fibonacciho čísla pomocí úloh

vektorizován. Tuto konstrukci je možné použít takto samostatně, nebo v kombinaci s `parallel for`. Můžeme ji také použít v kombinaci s redukcí. Vektorizaci je také možné provést manuálně použitím příslušných funkcí jazyka C++.

Úlohy

OpenMP také umožňuje používání úloh (tasks) [18]. Úloha je zjednodušeně řečeno část kódu, která se má vykonat. V programu můžeme úlohy vytvářet, ty pak mohou vytvářet další. Úlohy pak mohou být vykonány paralelně. Ve výpisu kódu 5.6 je pro příklad implementován výpočet Fibonacciho čísla, který je paralelizován pomocí úloh (funkce musí být spuštěna v paralelním bloku). Na šestém a osmém řádku se vytvoří nové úlohy, které je třeba provést – vypočtení nižších Fibonacciho čísel. Takto dojde k rekurzivnímu vytvoření určitého množství úloh, které se provedou paralelně. Na desátém řádku se čeká na dokončení vytvořených úloh, poté funkce vrátí vypočtenou hodnotu.

5.4 Výpočet zrychlení

Třída `Simulation` obsahuje pro výpočet zrychlení všech částic metodu `CalcAccelerations`. Ta volá metodu `CalcAccelerationsNoG` a výsledné hodnoty vynásobí gravitační konstantou, což je efektivnější, než jí násobit každé dílčí zrychlení. Ve zbytku této podkapitoly je tedy pojmem zrychlení myšleno zrychlení bez gravitační konstanty. Metoda `CalcAccelerationsNoG` podle nastavení simulace rozhodne, který algoritmus bude pro výpočet zrychlení použit, a zavolá příslušnou metodu.

5.4.1 Výpočet zrychlení mezi dvěma částicemi

Třída `Simulation` obsahuje statickou funkci `CalculateAccelerationNoG`. Ta má za úkol spočítat zrychlení jedné částice vůči jiné částici. Jejími vstupy jsou složky polohového vektoru primární částice (jejíž zrychlení počítáme) a sekundární částice (vůči níž zrychlení počítáme) a hmotnost sekundární částice. Výstupem funkce je vektor zrychlení působící na primární částici.

```
#pragma omp parallel for private(accx, accy, accz)
    schedule(runtime)
for (int i = 0; i < count; i++)
{
    FloatType totalAccX = 0;
    FloatType totalAccY = 0;
    FloatType totalAccZ = 0;

#pragma omp simd reduction(+: totalAccX, totalAccY, totalAccZ)
    for (int j = 0; j < count; j++)
    {
        CalculateAccelerationNoG(accx, accy, accz,
            positions->X(i), positions->Y(i), positions->Z(i),
            positions->X(j), positions->Y(j), positions->Z(j),
            masses[j]
        );

        totalAccX += accx;
        totalAccY += accy;
        totalAccZ += accz;
    }

    accelerations->X(i) = totalAccX;
    accelerations->Y(i) = totalAccY;
    accelerations->Z(i) = totalAccZ;
}
```

Výpis kódu 5.7: Implementace úplné přímé metody

5.4.2 Přímá metoda

Algoritmus přímé metody je implementován v metodě `CalcAccelerationsNoGDirect`. V této metodě se podle nastavení simulace rozhodne, zda se vypočtou všechna zrychlení, nebo se využije vztahu $\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji}$ a spočítá se jen polovina. Obě varianty mají paralelizovaný vnější cyklus. Podstatná část implementace úplné varianty je zobrazena ve výpisu kódu 5.7. Implementace poloviční varianty je složitější – při aktualizaci hodnot zrychlení by mohlo docházet k souběhu. Na vnějším cyklu je tedy pro hodnoty zrychlení použita vlastní implementace redukce – pro každé vlákno je vytvořena nová instance třídy `VectorCollection`, do které se přičítají zrychlení spočtené v příslušném vlákně a na konci se všechny hodnoty sečtou. Tím je oproti použití `atomic` i `critical` docíleno daleko nižší režie.

5.4.3 Barnesův-Hutův algoritmus

Barnesův-Hutův algoritmus je implementován v metodě `CalcAccelerationsNoGBarnesHut`. Ta využívá třídu `Octree`, která obaluje a spravuje stromovou strukturu částic tvořenou instancemi třídy `OctreeNode`.

Třída `Octree` obsahuje metodu `ConstructTree`, která podle nastavení simulace zavolá jednu ze tří metod pro samotné zkonstruování stromu zvoleným způsobem. Parametry těchto metod

```
1  for (int i = 0; i < subnodesCount; i++)
2  {
3  #pragma omp task firstprivate(i) if(this->particlesCount >
                        SimulationConfig::bhTreeConstrTaskMinParts)
4      {
5          subnodes[i]->CalculateProperties();
6      }
7  }
8  #pragma omp taskwait
9
10 for (int i = 0; i < subnodesCount; i++)
11 {
12     totalMass += subnodes[i]->totalMass;
13
14     centerOfMass.x += subnodes[i]->totalMass *
                          subnodes[i]->centerOfMass.x;
15     centerOfMass.y += subnodes[i]->totalMass *
                          subnodes[i]->centerOfMass.y;
16     centerOfMass.z += subnodes[i]->totalMass *
                          subnodes[i]->centerOfMass.z;
17 }
18
19 centerOfMass.x /= totalMass;
20 centerOfMass.y /= totalMass;
21 centerOfMass.z /= totalMass;
```

Výpis kódu 5.8: Výpočet vlastností buněk

jsou pozice všech částic a jejich hmotnosti. Metoda `ConstructTreeInserting` vytváří strom postupným vkládáním částic do kořenové buňky. Zbylé dvě metody vytváří strom rekurzivním dělením všech částic do osmi kategorií podle toho, do které buňky náleží. Tato varianta je implementována dvěma způsoby.

Prvním z nich je způsob, při kterém se spolu s pozicemi a hmotnostmi všech částic posílají hlouběji do stromu indexy částic, které přísluší do dané podbuňky. Tento způsob implementuje metoda `ConstructTreeSplittingIndex`. Druhý způsob seřadí částice a jejich hmotnosti podle toho, do které podbuňky patří. Hlouběji do stromu pak posílá seřazené částice a rozsah, kde v kolekci se pozice a hmotnosti příslušných částic nacházejí. Řazením se sice provede práce navíc, může se to však vyplatit z důvodu lepší lokality dat v cache paměti a možnosti využití vektorové jednotky u některých cyklů. Tuto variantu implementuje metoda `ConstructTreeSplittingSort`.

První způsob, vkládání, není v programu paralelizován. Obě varianty druhého způsobu, dělení částic do kategorií, jsou paralelizovány pomocí úloh. Po roztřídění částic do kategorií se vytvoří několik úloh, přičemž každá z nich rekurzivně vytváří jednu z podbuněk.

Po zkonstruování stromu se rekurzivně vypočtou vlastnosti všech uzlů, tj. jejich celková hmotnost a hmotný střed, pomocí metody `CalculateProperties`. Tato metoda je také paralelizovaná pomocí úloh, viz výpis kódu 5.8, ve kterém je podstatná část této metody zobrazena.

```
#pragma omp parallel for schedule(runtime)
    for (int i = 0; i < partCount; i++)
    {
        root->CalcParticleAccelerationNoG(
            accelers->X(i), accelers->Y(i), accelers->Z(i),
            positions->X(i), positions->Y(i), positions->Z(i));
    }
```

Výpis kódu 5.9: Výpočet zrychlení Barnesovým-Hutovým algoritmem

Samotný výpočet zrychlení částic provádí metoda `CalcParticlesAccelerationsNoG`. Ta cyklem projde všechny částice a pro každou zavolá metodu `CalcParticleAccelerationNoG` na kořeni stromu, která rekurzivně spočítá zrychlení dané částice. Cyklus iterující přes všechny částice je paralelizován, viz výpis kódu 5.9.

5.5 Vstupní a výstupní soubory

Ze vstupních souborů se načítají počáteční podmínky a do výstupních souborů jsou ukládány vypočtené polohy a rychlosti částic v jednotlivých krocích simulace. Soubory jsou dvojího typu – buď textové, nebo binární. Vstupní i výstupní soubory mají stejný formát a obsahují hmotnosti, polohy a rychlosti všech částic. Textové soubory mají tyto informace pro dobrou čitelnost uloženy stylem AoS, binární jsou ve formátu SoA. Typ souboru programy poznají podle jeho přípony – `.pst` značí textový soubor, `.psf` a `.psd` jsou binární soubory, v nichž je použit datový typ `float`, resp. `double`.

5.6 Ovládání programu

Všechny programy se spouští přes příkazovou řádku a jejich vstupy jsou pouze argumenty příkazové řádky, popřípadě jimi specifikované vstupní a výstupní soubory. Programy nenačítají data ze standardního vstupu a na standardní výstup vypisují některé informace o běhu.

Vstupními argumenty programu `UniverseGenerator` jsou cesta k výstupnímu souboru, počet částic, celková hmotnost systému, model, jeho parametr a hodnota pro inicializaci generátoru náhodných čísel. `UniverseModifier` v argumentech požaduje cestu k výstupnímu a vstupním souborům a typ s parametry požadované modifikace systému. Parametry programu `Converter` upřesňují cestu ke vstupnímu a výstupnímu souboru. V případě konverze do formátu `.vtk` lze specifikovat maximální počet snímků animace. Programu `ErrorAnalyzer` lze specifikovat druh chyby, která se počítá a cesty ke výstupnímu a vstupním souborům. Všechny programy při zadání prvního argumentu `-h` stručně vypíší jejich použití.

Program `ParticleSimulation` má mnohem více možností nastavení a tedy složitější strukturu vstupních argumentů. V tabulce 5.1 jsou rozepsány všechny argumenty, jejich význam, případný parametr a výchozí hodnota.

Většina argumentů je nepovinných a má výchozí hodnotu, nutné je však zadat cestu ke vstupnímu souboru a cestu k výstupnímu souboru nebo `--nooutput`. Vstupní a výstupní soubor není nutné označovat přepínači `-i`, resp. `-o`. První argument, který není označen přepínačem specifikující jeho význam, je interpretován jako vstupní soubor, druhý jako výstupní soubor.

Tabulka 5.1: Vstupní argumenty programu ParticleSimulation

Argument	Parametr	Výchozí	Význam
Vstupní a výstupní soubory			
-i, --input	ANO	–	cesta ke vstupnímu souboru
-o, --output	ANO	–	cesta k výstupnímu souboru
--nooutput	NE	NE	výstup simulace nebude uložen
Parametry simulace			
-t, --time	ANO	1	celkový simulační čas v sekundách
-s, --steps	ANO	0+1	počet kroků simulace
--soft, --softening	ANO	1	zjemňovací parametr ϵ
Numerické řešiče			
--euler	NE	ANO	standardní Eulerova metoda
--verlet, --leapfrog	NE	NE	rychlostní Verletova metoda
--rungekutta	NE	NE	metoda Rungeho-Kutta čtvrtého řádu
Algoritmy výpočtu zrychlení			
--direct	NE	ANO	přímá metoda particle-particle
--full	NE	ANO	úplná varianta přímé metody
--half	NE	NE	poloviční varianta přímé metody
--barneshut	NE	NE	Barnesův-Hutův algoritmus
--insert	NE	NE	konstrukce BH stromu vkládáním
--splitindex	NE	NE	konstrukce BH stromu dělením s indexy
--splitsort	NE	ANO	konstrukce BH stromu řadícím dělením
--theta	ANO	1	hodnota θ u BH algoritmu
Paralelizace			
--threads	ANO	1	počet spuštěných vláken
--schedule	ANO	auto	scheduling některých paralelizovaných cyklů
--chunksize	ANO	-1	velikost dávky některých paralelizovaných cyklů
--taskminpconstr	ANO	128	minimální počet částic pro vytvoření nové úlohy při konstrukci stromu
--taskminppcalc	ANO	256	minimální počet částic pro vytvoření nové úlohy při výpočtu vlastností stromu

Parametr `-s` nebo `--steps` následuje číslo určující počet simulačních kroků. Je možno zadat například `-s 2+10`, kdy se provede celkem 12 kroků, přičemž první dva jsou inicializační – nezapočítávají se do časových statistik a na výstupu se nijak neprojeví. Při zadání jen jednoho čísla se započítávají časy všech kroků. Tato funkčnost je implementována pro účely testování rychlosti běhu programu – prvotní iterace mají z důvodu inicializace paměti, vláken a podobně větší časový overhead.

6 Numerické experimenty

Numerické experimenty probíhaly na výpočetním uzlu clusteru Salomon v Národním superpočítačovém centru IT4Innovations. Jeden uzel je vybaven dvěma 12-jádrovými procesory Intel Xeon E5-2680v3 (architektura Haswell) a 128 GB paměti RAM. Pro překlad programů spouštěných na clusteru byl použit Intel C++ Compiler verze 17.0.1. Programy byly překládány s použitím přepínačů `-qopenmp`, `-O3` a `-ipo`, tj. byla zapnuta podpora OpenMP, použita nejvyšší možná optimalizace a proběhly optimalizace napříč funkcemi a moduly.

Tabulky a grafy numerických experimentů pro stručnost neobsahují všechna naměřená data. Ta je možné nalézt v příloze této práce. U všech experimentů měřících čas běhu byl čas měřen jako aritmetický průměr 20 kroků simulace Eulerovou metodou, kterým předcházely dva inicializační. K měření času byla použita třída `std::chrono::steady_clock`.

6.1 Počáteční podmínky, Plummerův model

Pro testování programu potřebujeme nějaká vstupní data – počáteční podmínky systému částic. K jejich vygenerování použijeme zejména Plummerův model [5]. Ten představuje sféricky symetrickou hvězdokupu, jejíž hustota je nejvyšší v jejím středu a s rostoucí vzdáleností se snižuje. Protože je systém sféricky symetrický, počítáme jeho vlastnosti v závislosti na vzdálenosti r od středu systému, za který považujeme bod $(0, 0, 0)$. Plummerův model je sestaven tak, aby byl již zpočátku v dynamické rovnováze, tzn. že částice sice jsou v pohybu, vnější struktura systému (hustota částic v jednotlivých částech) se ale nemění.

Systém částic v dynamické rovnováze je definován dvojicí gravitační potenciál-hustota. Gravitační potenciál (gravitační potenciální energie částice o jednotkové hmotnosti) ve vzdálenosti r od středu hvězdokupy podle Plummerova modelu je

$$\Phi(r) = -G\frac{M}{\sqrt{r^2 + a^2}}, \tag{6.1}$$

kde M je hmotnost celé hvězdokupy, a je parametr ovlivňující velikost jejího jádra a G je gravitační konstanta. Z potenciálu lze odvodit hustotu částic v závislosti na vzdálenosti od středu hvězdokupy,

$$\rho(r) = \frac{3M}{4\pi}\frac{a^2}{(r^2 + a^2)^{5/2}}. \tag{6.2}$$

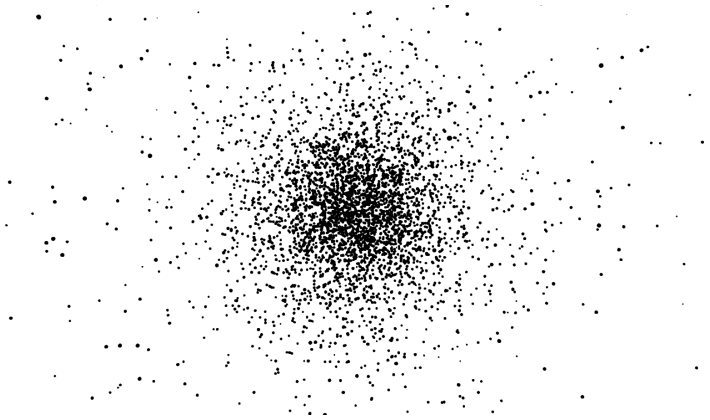
Abychom docílili požadované hustoty, generujeme vzdálenost každé částice od středu systému předpisem

$$r = \frac{a}{\sqrt{\mu^{-2/3} - 1}}, \tag{6.3}$$

kde μ volíme rovnoměrně náhodně z intervalu $\langle 0, 1\rangle$. Následně rovnoměrně náhodně vygenerujeme bod na povrchu koule o poloměru r. Tento bod je pozicí nově vygenerované částice. Ta se s největší pravděpodobností bude nacházet v blízkosti středu systému, může však být vygenerována jakkoli daleko (když pomineme omezení rozsahu čísel s plovoucí řádovou čárkou).

Rychlost částice generujeme v polárních souřadnicích. Její velikost musí být v rozmezí $0 \leq v \leq v_e(r)$, kde

$$v_e(r) = \frac{\sqrt{2GM}}{(r^2 + a^2)^{1/4}} \tag{6.4}$$



Obrázek 6.1: Jádro hvězdokupy vygenerované podle Plummerova modelu

je úniková rychlost ve vzdálenosti r od středu systému. Označme $q = \frac{v}{v_e(r)}$, potom $q \in \langle 0, 1\rangle$. Podle Plummerova modelu má být pravděpodobnostní rozložení hodnoty q úměrné funkci

$$h(q) = q^2(1 - q^2)^{7/2}. \tag{6.5}$$

Hodnotu q pak náhodně vygenerujeme s použitím techniky odmítnutí (rejection sampling). Vygenerujeme rovnoměrně náhodně čísla $q \in \langle 0, 1\rangle$, $y \in \langle 0,\ 0{,}1\rangle$ (maximum funkce $h(q)$ na intervalu $\langle 0, 1\rangle$ je přibližně 0,092). Pokud je splněna nerovnost $y \leq h(q)$, hodnotu q přijmeme a spočítáme rychlost $v = qv_e(r)$. Pokud nerovnost splněna není, hodnotu q odmítneme a opakujeme. Tím dostaneme hodnoty rychlostí se správným rozložením.

Všechny částice mají stejnou hmotnost $m = \frac{M}{n}$. Obvyklá hmotnost sférické hvězdokupy se pohybuje v rozmezí 10^4 až 10^6 slunečních hmotností [9], přičemž hmotnost slunce je $M_\odot \approx 2 \cdot 10^{30}$ kg. Poloměr jádra hvězdokupy pak bývá okolo 1 pc $\approx 3{,}1 \cdot 10^{16}$ m. V literatuře se však často setkáváme s volbou $G = M = a = 1$, což může zjednodušit výpočty a snížit nároky na rozsah čísel s plovoucí řádovou čárkou. V této práci se však tohoto zvyku příliš nedržíme. Podrobnější rozbor generování částic podle Plummerova modelu přesahuje rozsah této práce a čtenář jej může nalézt například v [5]. Na obrázku 6.1 je pro ukázku vizualizováno jádro hvězdokupy vygenerované podle Plummerova modelu.

Pro simulaci je třeba zvolit hodnotu zjemňovacího parametru ϵ. Ta má být přibližně rovna polovině střední vzdálenosti částic v nejhustší části systému [17], kterou je jádro hvězdokupy o hustotě $\rho(0) = \frac{3M}{4\pi a^3}$. Při znalosti hmotnosti jedné částice $m = \frac{M}{n}$ jsme schopni spočítat objem, který přibližně zabírá jedna částice, $V = \frac{m}{\rho(0)} = \frac{4\pi}{3}\frac{a^3}{n}$. Z něj spočítáme poloměr příslušné koule, který je dvojnásobkem střední vzdálenosti částic. Zjemňovací parametr pro Plummerův model tedy volíme přibližně $\epsilon = \frac{a}{\sqrt[3]{n}}$.

Délka časového kroku by měla být taková, že částice v průměru za jeden časový krok překoná dráhu menší než $\epsilon/2$ [17]. Nejvyšší rychlost, která může být částici přiřazena, je $v_e(0) = \sqrt{2GM/a}$. Spokojíme se se střední hodnotou rychlosti $v_e(0)/2$, z čehož dostaneme, že by délka časového kroku Δt měla být menší než $\frac{a}{\sqrt[3]{n}}\sqrt{\frac{a}{2GM}}$.

Programem lze vygenerovat také částice s rovnoměrným rozložením v kouli s konstantní hustotou. Všechny částice mají stejnou hmotnost a je jim přiřazena nulová rychlost.

6.2 Chyby numerických metod řešení počátečních úloh

Celková energie, hybnost a moment hybnosti jsou konstantními charakteristikami systému částic, tj. hodnoty těchto veličin se v čase nemění [7]. Toho můžeme využít pro kontrolu přesnosti simulací, ve kterých tyto veličiny konstantní nezůstanou, a to z důvodu výskytu chyb způsobených použitou numerickou metodou, nepřesným výpočtem sil nebo omezením přesnosti čísel s plovoucí řádovou čárkou. Pro měření přesnosti numerických řešičů diferenciálních rovnic použijeme celkovou energii systému. Tu spočítáme jako součet kinetické a potenciální energie všech částic [7],

$$E = E_K + E_P = \sum_{i=1}^{n} \frac{1}{2} m_i \|\boldsymbol{v}_i\|^2 - \sum_{i=1}^{n} \sum_{j=1}^{i-1} G \frac{m_i m_j}{\|\boldsymbol{r}_i - \boldsymbol{r}_j\|}. \tag{6.6}$$

Relativní chybu energie v k-tém kroku pak spočteme vzorcem

$$e_{E,k} = \left| \frac{E_k - E_0}{E_0} \right|, \tag{6.7}$$

kde E_k je celková energie v k-tém kroku a E_0 je energie systému v počátečním kroku.

Výpočet celkové potenciální energie systému má kvadratickou složitost, tato metoda tedy není vhodná pro kontrolu přesnosti simulace systémů s větším počtem částic. Pro ty lze využít zbylé konstantní charakteristiky systému, které lze spočítat s lineární složitostí. Nutno zmínit, že velmi malá odchylka v těchto charakteristikách nemusí nutně znamenat realistickou simulaci. Pokud například numerický řešič polohu ani rychlost částic nijak nemění, všechny vlastnosti jsou přesně zachovány, systém se však nechová reálně.

Pro kontrolu přesnosti numerických řešičů simulujeme systém se dvěma částicemi, které se navzájem obíhají po eliptických drahách. Celkový čas simulace je zvolen tak, že za něj částice provedou přes 6 oběhů. Volíme zanedbatelnou hodnotu zjemňovacího parametru $\epsilon = 10^{-12}$, resp. $\epsilon = 10^{-90}$ pro datový typ float, resp. double při počáteční vzdálenosti částic jeden metr. Simulaci pro každý numerický řešič provedeme pro celkový počet 10^2 až 10^7 kroků, přičemž celkový simulační čas ponecháme stejný. V tabulce 6.1 jsou vypsány maximální relativní chyby energie a odhady řádu konvergence pro různé numerické metody a celkové počty kroků při použití datového typu double. Odpovídající závislost je graficky vyobrazena na obrázku 6.2, plnou čarou pro typ double a čárkovanou pro float. Na obrázku 6.3 je zobrazen vývoj chyby standardní Eulerovy metody v závislosti na kroku simulace.

Z grafu 6.2 je patrné, že chyba standardní Eulerovy metody klesá s rostoucím počtem kroků nejpomaleji. Chyba Rungeho-Kuttovy metody čtvrtého řádu klesá nejrychleji. Po dosažení přesnosti přibližně 10^{-5} pro typ float, resp. 10^{-13} pro double se začne výrazně projevovat omezení přesnosti výpočtů v pohyblivé řádové čárce. Odhady řádu konvergence přibližně odpovídají řádům příslušných metod, u Rungeho-Kuttovy metody čtvrtého řádu je odhad řádu dokonce vyšší.

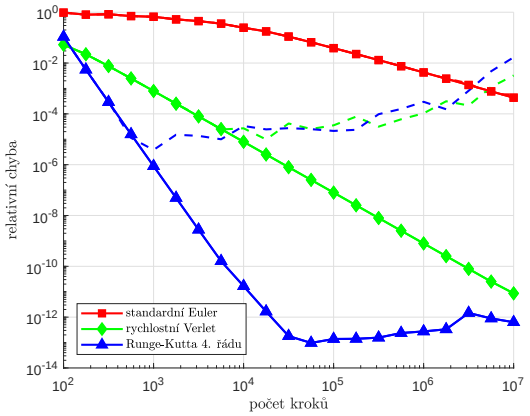
6.3 Chyba Barnesova-Hutova algoritmu

Chybu způsobenou algoritmem výpočtu sil zjistíme porovnáním vypočtených vektorů síly s vektory síly spočtenými přímou metodou, které považujeme za správné. Absolutní chybu spočítáme jako normu rozdílu těchto dvou vektorů. Relativní chyba je dána zlomkem

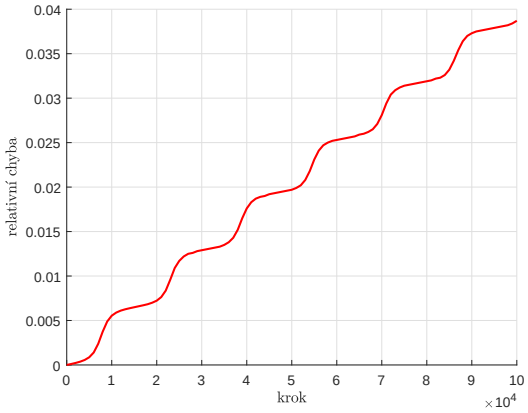
$$e_F = \frac{\|\boldsymbol{F}_b - \boldsymbol{F}_p\|}{\|\boldsymbol{F}_p\|}, \tag{6.8}$$

Tabulka 6.1: Relativní chyba energie a odhad řádu konvergence v závislosti na počtu kroků

počet kroků		10^2	10^3	10^4	10^5	10^6	10^7
standardní Euler	chyba	9.64e−01	6.62e−01	2.45e−01	3.87e−02	4.27e−03	4.33e−04
	o.ř.k.	–	0.16	0.43	0.80	0.95	0.99
rychlostní Verlet	chyba	5.29e−02	7.92e−04	7.96e−06	7.96e−08	7.96e−10	8.63e−12
	o.ř.k.	–	1.82	1.99	2.00	2.00	1.96
Runge-Kutta 4. řádu	chyba	1.07e−01	8.75e−07	1.66e−11	1.38e−13	2.74e−13	6.32e−13
	o.ř.k.	–	5.08	4.72	2.08	−0.29	−0.36



Obrázek 6.2: Závislost relativní chyby na celkovém počtu kroků



Obrázek 6.3: Závislost relativní chyby standardní Eulerovy metody na kroku simulace

kde $\boldsymbol{F}_p$ je vektor síly spočtený přímou metodou a $\boldsymbol{F}_b$ je vektor síly vypočtený Barnesovým-Hutovým algoritmem.

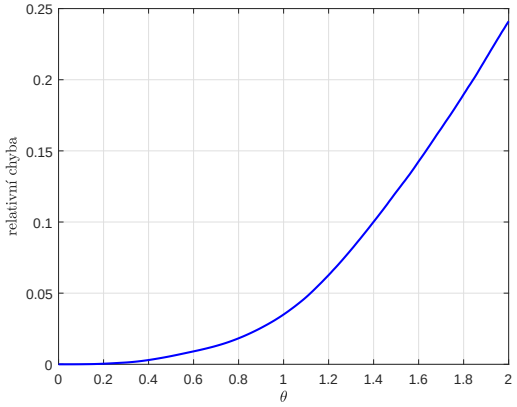
Souhrnnou relativní chybu celého systému počítáme jako aritmetický průměr relativních chyb sil působících na jednotlivé částice. Chyby měříme na počtu 10^5 částic generovaných podle Plummerova modelu. Simulace provádíme s dvojitou přesností výpočtů v pohyblivé řádové čárce pro hodnoty $0 \leq \theta \leq 2$. V tabulce 6.2 jsou vypsány celkové relativní chyby pro různé hodnoty θ, na obrázku 6.4 je odpovídající závislost graficky vyobrazena. Při použití obvyklých hodnot $0{,}5 \leq \theta \leq 1$ se průměrná relativní chyba pohybuje mezi 0,57 a 3,5 procenty. Na obrázku 6.5 je vykreslena závislost doby samotného výpočtu sil na volbě parametru θ.

6.4 AoS a SoA, vektorizace

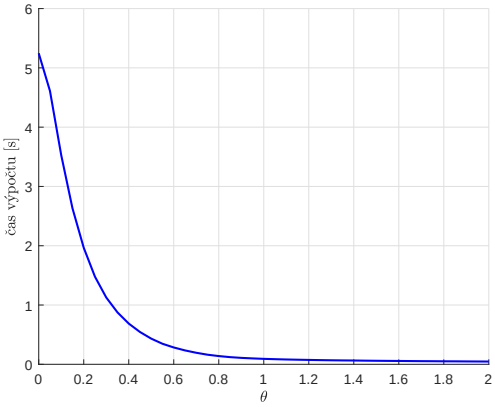
V kapitole 5.2 bylo zmíněno, že uspořádání paměti stylem SoA by mělo být oproti AoS časově efektivnější. Použití vektorové jednotky by také mělo výpočty zrychlit. Zmíněné domněnky jsme tedy otestovali. Testování probíhalo na jednom vlákně pro obě uspořádání paměti, bez vektori-

Tabulka 6.2: Relativní chyba síly v závislosti na parametru θ

θ	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
chyba	0	0.0001	0.0004	0.0012	0.0030	0.0057	0.0090	0.0129
θ	0.8	0.9	1	1.2	1.4	1.6	1.8	2
chyba	0.0182	0.0256	0.0350	0.0627	0.1001	0.1426	0.1897	0.2410



Obrázek 6.4: Relativní chyba síly v závislosti na parametru θ



Obrázek 6.5: Závislost času výpočtu sil na parametru θ

zace a pro vektorové instrukční sady SSE4.2 a AVX2[6] s šířkami vektorového registru 128, resp. 256 bitů. Jedna hodnota datového typu float má velikost 32 bitů, u typu double je to 64 bitů, Zrychlení by mělo být přibližně rovno počtu hodnot, které je vektorový registr schopen pojmout.

V tabulce 6.3 jsou vypsány časy výpočtu sil v systému s $2^{14} = 16384$ částicemi úplnou přímou metodou. U Barnesova-Hutova algoritmu použití vektorizace ani uspořádání dat v paměti na čas běhu výrazný vliv neměly, věnujeme se tedy jen přímé metodě.

Z tabulky lze vyčíst, že uspořádání paměti stylem SoA je vždy výhodnější než AoS, což souhlasí s očekáváním. Významný rozdíl se však projevil pouze při použití datového typu float, u typu double je rozdíl téměř nulový.

Vektorová jednotka zrychlí výpočty při použití datového typu float a uspořádání SoA podle očekávání, pro SSE4.2 čtyřikrát a pro AVX2 dokonce více než osmkrát. Při použití AoS je zrychlení nižší, ale srovnatelné. U typu double jsou výpočty při použití SSE4.2 zrychleny podle očekávání dvakrát, zvláštností je však to, že zrychlení AVX2 je oproti SSE4.2 mírně nižší, přestože by podle očekávání mělo být dvojnásobné. Nejvýhodnější je tedy použití uspořádání paměti SoA a vektorové instrukční sady AVX2, čehož se v dalších testech budeme držet.

[6] tedy s přepínači kompilátoru `-no-vec -no-simd -qno-openmp-simd`, resp. `-xSSE4.2`, resp. `-xCORE-AVX2`

Tabulka 6.3: Časy výpočtu sil pro různá uspořádání paměti a vektorové instrukční sady

	float			double		
	AoS	SoA	AoS/SoA	AoS	SoA	AoS/SoA
bez vektorizace	1574.28	1544.18	1.019	2298.53	2296.47	1.001
SSE4.2	477.06	385.43	1.237	1160.48	1148.71	1.010
AVX2	261.41	184.53	1.416	1226.99	1219.56	1.006
zrychlení SSE4.2	3.299	4.006		1.980	1.999	
zrychlení AVX2	6.022	8.367		1.873	1.883	

6.5 Optimální plánování a omezení vytváření úloh

Na dobu běhu paralelních výpočtů má nemalý vliv typ použitého plánování a velikost dávky. Není vhodné spustit výpočet sil 1000 částic na 24 vláknech při statickém plánování a velikosti dávky 1000 – výpočet prakticky poběží pouze na jednom vlákně. Také není optimální zvolit pro dynamické plánování dávku velikosti 1, režie plánování bude v tom případě vysoká. Testováním různých plánování a velikostí dávky na různém počtu částic jsme zjistili optimální plánování a velikost dávky pro jednotlivé paralelizované cykly v programu. Čas výpočtu se ale při použití automatického plánování ve všech případech téměř shodoval s experimentálně zjištěným optimálním plánováním. OpenMP tedy dokázalo pro každý cyklus vybrat optimální plánování. Budeme tedy nadále používat automatické plánování.

Vytváření stromové struktury metodou dělení je paralelizované pomocí úloh. Pro každý uzel stromu je vytvořena jedna úloha, která částice náležící do dané buňky roztřídí do příslušných podbuněk a pro jejich vytváření vytvoří další úlohy. Pokud bychom vytvářeli úlohu pro každou novou buňku, byla by režie spojená s vytvářením úloh významná. Nastavíme tedy limit na počet částic v buňce, pod který se nové úlohy nebudou vytvářet a vytváření všech podbuněk bude probíhat ve stejné úloze. Pokud bychom ale nastavili limit příliš vysoko, například při 1000 částicích bychom zvolili limit 1000, nová úloha by se nikdy nevytvořila, celý proces vytváření stromu by se tedy provedl sekvenčně. Mezi těmito dvěma extrémy musí existovat optimální hodnota omezení. To platí nejen pro proces vytváření stromu, ale také pro jeho procházení při výpočtu vlastností uzlů. Experimenty bylo zjištěno, že optimální omezení pro vytváření stromu je okolo 128 částic, pro jeho procházení pak 256. Výrazná závislost optimálního omezení na celkovém počtu částic nebyla zjištěna.

6.6 Paralelní škálovatelnost algoritmů

Spuštěním výpočtů na více vláknech by se měl čas potřebný k dokončení zkrátit a program by se měl zrychlit. Zrychlení způsobené paralelizmem počítáme jako zlomek $S = \frac{T_1}{T_N}$, kde T_1 je doba trvání výpočtu na jednom vlákně a T_N je čas potřebný k dokončení na N vláknech. Ideální zrychlení je rovno počtu vláken na kterých je program spuštěn, z důvodu paralelizační režie a neparalelizovatelných částí kódu však bývá nižší. Efektivita $\frac{S}{N}$ pak představuje míru využití potenciálu vícejádrového systému. Její ideální hodnota je 1, prakticky však bývá nižší [3].

Nejvyšší možné paralelní zrychlení (při zanedbání paralelizační režie) popisuje Amdahlův zákon [3]. Jestliže sekvenční neparalelizovaná část kódu při sekvenčním běhu zabírá část s času běhu programu (paralelizovaná část tedy představuje část $1-s$), pak zrychlení na N vláknech je

$$S = \frac{1}{s + \frac{1-s}{N}}. \tag{6.9}$$

Nejvyšší možné zrychlení je tedy $\frac{1}{s}$. Například při 10 % neparalelizovaného kódu jsme schopni výpočty na libovolně velkém počtu vláken zrychlit maximálně desetkrát.

V této podkapitole otestujeme paralelní škálovatelnost, zrychlení a efektivitu obou algoritmů výpočtu sil, jejich částí a variant. Používáme datový typ double, paralelní zrychlení se od typu float ve většině případů výrazně nelišilo. Je zajištěno, že je při nízkém počtu vláken používán pouze jeden procesor, druhý se používá až od 13 vláken výše. Vlákna jsou od jejich spuštění připnuta na jedno jádro, mezi jádry tedy nemigrují, čímž se zlepší lokalita dat v paměti.

6.6.1 Přímá metoda

Přímá metoda je v programu implementována dvěma variantami. Úplná varianta počítá všech n^2 sil, zatímco poloviční využívá vztahu $\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji}$ a provádí pouze $\frac{1}{2}n(n-1)$ výpočtů sil. U poloviční varianty je ovšem třeba, byť jednoduše, spočítat a přičíst sílu působící na druhou částici. Také je třeba vyvarovat se souběhu, což může přinést další zpomalení. Je tedy otázkou, zda se vůbec vyplatí používat poloviční variantu oproti úplné. Také zjistíme, jak obě varianty paralelně škálují.

V tabulce 6.4 jsou vypsány časy výpočtu sil oběma variantami přímé metody pro různé počty vláken. Je spočítáno časové zlepšení poloviční varianty oproti úplné, zrychlení způsobené paralelizmem a efektivita. Na obrázku 6.6 je graf zachycující dobu běhu obou variant přímé metody v závislosti na počtu vláken, na obrázku 6.7 je pak vykreslena závislost efektivity na počtu vláken. Testy probíhaly na systému $2^{15} = 32768$ částic.

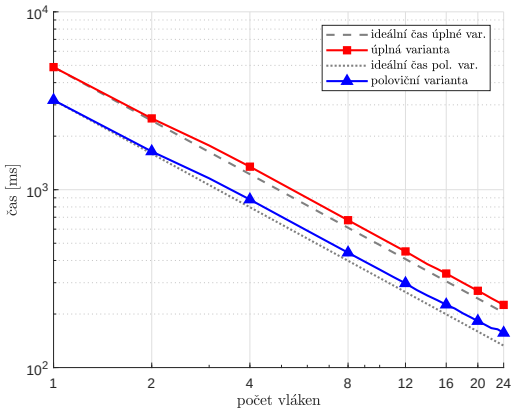
Z grafů a tabulky je patrné, že použití poloviční varianty je časově výhodnější. Obě varianty škálují téměř ideálně, úplná varianta však škáluje lépe. Výhoda poloviční varianty se tedy s rostoucím počtem vláken mírně snižuje. Pro datový typ float jsme provedli podobné měření, v časech výpočtu mezi úplnou a poloviční variantou jsme však významný rozdíl nezaznamenali. Poloviční varianta je implementována za použití vlastní implementace redukce, s použitím atomických příkazů byla poloviční varianta oproti úplné 6-krát pomalejší.

6.6.2 Konstrukce stromu Barnesova-Hutova algoritmu

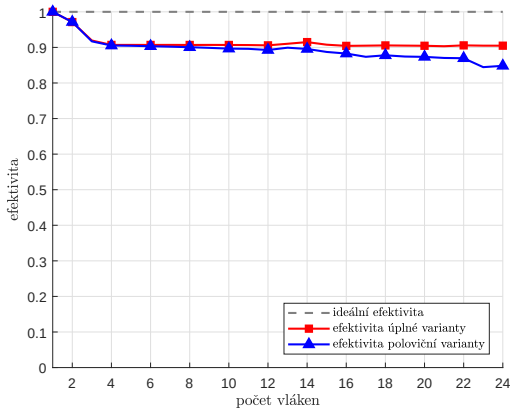
V předchozích kapitolách byly zmíněny tři varianty vytváření stromu pro Barnesův-Hutův algoritmus. Vkládání jedné částice po druhé do kořenové buňky a dva způsoby dělení souboru částic do příslušných podbuněk a jejich rekurzivní vytváření. Varianta vkládání není jednoduše paralelizovatelná, zbylé dvě jsou paralelizovány pomocí úloh – po roztřídění částic do kategorií mohou být podbuňky vytvářeny paralelně nezávisle na sobě. Připomeňme, že indexovací varianta konstrukce stromu dělením posílá hlouběji do stromu indexy částic příslušících do dané podbuňky, zatímco řadící varianta částice seřadí podle toho, do které podbuňky patří. Řazení částic sice zabere nějaký čas, investice se však může vyplatit z důvodu lepší lokality dat v paměti. Testováním byly varianty porovnány a byla zjištěna jejich paralelní škálovatelnost.

Tabulka 6.4: Časy výpočtu sil přímou metodou na různých počtech vláken

počet vláken	1	2	4	8	12	16	24
čas úplné varianty [ms]	4888.17	2516.51	1347.32	673.74	449.71	337.83	225.09
čas poloviční varianty [ms]	3188.13	1640.88	880.42	442.52	297.61	225.72	156.60
porovnání variant	1.5332	1.5336	1.5303	1.5224	1.5110	1.4966	1.4373
zrychlení úplné varianty	1	1.94	3.62	7.25	10.86	14.46	21.71
zrychlení poloviční var.	1	1.94	3.62	7.20	10.71	14.12	20.35
efektivita úplné var. [%]	100	97.12	90.70	90.69	90.57	90.43	90.48
efektivita poloviční v. [%]	100	97.14	90.52	90.05	89.27	88.27	84.82



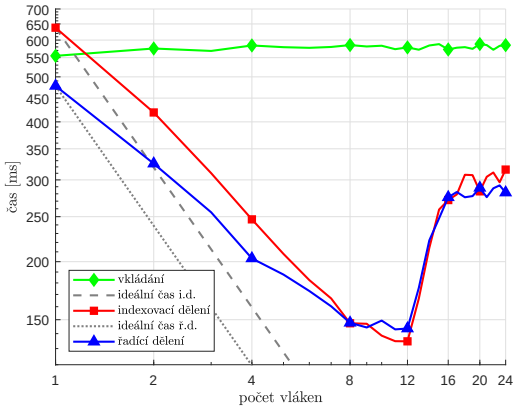
Obrázek 6.6: Závislost času výpočtu sil přímou metodou na počtu vláken



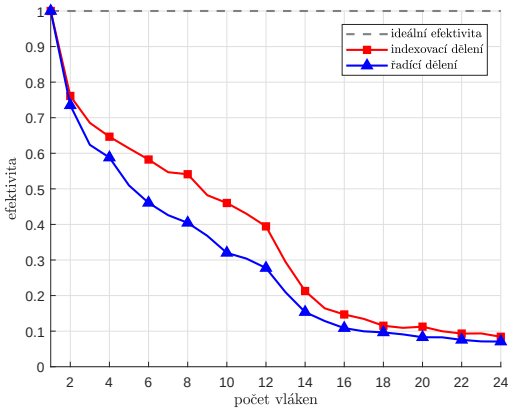
Obrázek 6.7: Efektivita výpočtu sil přímou metodou v závislosti na počtu vláken

Tabulka 6.5: Časy konstrukce stromu na různém počtu vláken

počet vláken	1	2	4	8	12	16	24
vkládání [ms]	554.87	575.45	584.19	585.31	578.58	572.69	585.08
indexovací dělení [ms]	637.97	419.07	246.68	147.37	134.75	271.46	315.75
řadící dělení [ms]	478.04	325.21	203.26	147.73	143.57	275.23	281.72
zrychlení indexovacího dělení	1	1.5223	2.5861	4.3288	4.7342	2.3500	2.0205
zrychlení řadícího dělení	1	1.4699	2.3518	3.2359	3.3294	1.7368	1.6968
efektivita index. dělení [%]	100	76.11	64.65	54.11	39.45	14.68	8.41
efektivita řadícího dělení [%]	100	73.49	58.79	40.44	27.74	10.85	7.07



Obrázek 6.8: Závislost doby konstrukce stromu na počtu vláken



Obrázek 6.9: Efektivita variant konstrukce stromu

V tabulce 6.5 jsou pro každou variantu konstrukce stromu vypsány doby výpočtu na různých počtech vláken. Je spočítáno zrychlení způsobené paralelizmem a efektivita obou variant dělení. Na obrázcích 6.8 a 6.9 jsou tyto hodnoty zobrazeny graficky.

Na méně než osmi vláknech je nejvýhodnější vytvářet strom variantou řadícího dělení, od osmi vláken výše jsou varianty dělení srovnatelné. Varianta vkládání je téměř vždy nejpomalejší, protože není paralelizována. Obě varianty dělení škálují relativně dobře do 12 vláken. Od 13 vláken se výpočty provádějí na obou procesorech a u obou variant dojde k tzv. NUMA efektu – z důvodu použití dvou procesorů se zhorší podmínky přístupu do paměti, čímž se sníží rychlost výpočtů. Pro optimální dobu konstrukce stromu tedy omezíme maximální počet vláken pro tento proces na 12.

Testování probíhalo na počtu $2^{20} \approx 10^6$ částic generovaných podle Plummerova modelu. Konstrukce stromu byla u rovnoměrného rozložení částic přibližně o 15 % rychlejší. Strom vytvořený z částic generovaných podle Plummerova modelu je totiž méně vyvážený a v nejhustších částech systému je hloubka uzlů větší. Špatná paralelní škálovatelnost konstrukce stromu dělením je nejspíše způsobena neparalelizovatelnými sekcemi kódu, který třídí částice do podbuněk. Třídění částic v kořenové buňce tedy probíhá pouze na jednom vlákně, což spolu s faktem, že kořenová buňka obsahuje nejvíce částic, částečně vysvětluje špatnou škálovatelnost.

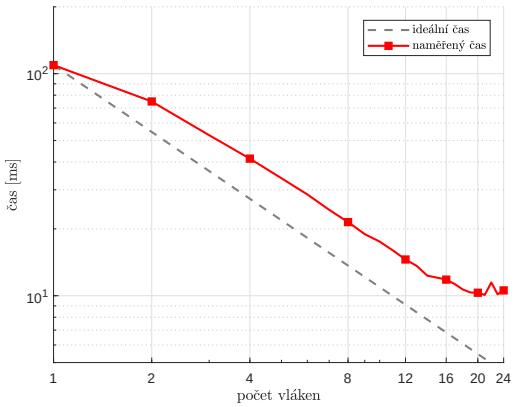
6.6.3 Výpočet vlastností buněk ve stromu

Výpočet vlastností stromu používaného v Barnesově-Hutově algoritmu je také paralelizován pomocí úloh. Při výpočtu vlastností buňky se nejprve vytvoří několik úloh pro výpočet vlastností jejích podbuněk. Poté, co se vytvořené úlohy dokončí, se z vlastností podbuněk vypočtou vlastnosti dané buňky.

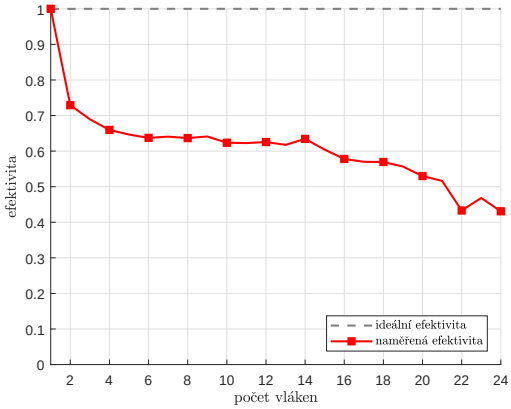
V tabulce 6.6 a obrázcích 6.10 a 6.11 je vidět, že výpočet vlastností buněk ve stromu neškáluje ideálně. Efektivita dosahuje do 12 vláken přibližně dvou třetin, poté ale z důvodu použití druhého procesoru začne výrazněji klesat. Na rozdíl od konstrukce stromu se však časově vyplatí

Tabulka 6.6: Časy výpočtu vlastností buněk na různém počtu vláken

počet vláken	1	2	4	8	12	16	24
čas [ms]	109.29	74.94	41.42	21.45	14.55	11.81	10.56
zrychlení	1	1.45	2.63	5.09	7.50	9.24	10.34
efektivita [%]	100	72.92	65.95	63.66	62.55	57.80	43.08



Obrázek 6.10: Závislost času výpočtu vlastností buněk na počtu vláken



Obrázek 6.11: Efektivita výpočtu vlastností buněk v závislosti na počtu vláken

provést výpočet na větším počtu vláken. Testování bylo prováděno na systému $2^{20} \approx 10^6$ částic generovaných podle Plummerova modelu. Rozdíl oproti rovnoměrnému rozložení je minimální.

6.6.4 Výpočet sil Barnesovým-Hutovým algoritmem

Samotný výpočet sil pomocí Barnesova-Hutova algoritmu je paralelizován relativně jednoduše. Paralelizován je cyklus iterující přes všechny částice, v jehož těle je volána metoda výpočtu síly na kořeni stromu. Lze tedy očekávat, že výpočet sil bude paralelně škálovat velmi dobře.

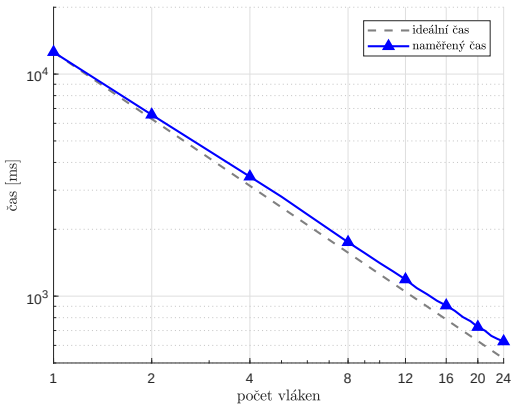
Z obrázků 6.12 a 6.13 a tabulky 6.7 lze usoudit, že výpočet sil dle očekávání škáluje téměř ideálně. Testování probíhalo na systému $2^{20} \approx 10^6$ částic generovaných podle Plummerova modelu. Výpočet sil nad systémem s rovnoměrným rozložením částic probíhal více než dvakrát rychleji, důvodem je již dříve zmíněná lepší vyváženost stromu a menší hloubka většiny uzlů. Paralelní zrychlení na rovnoměrném modelu bylo s Plummerovým modelem srovnatelné. Při testech byla volena hodnota parametru přesnosti $\theta = 1$.

6.6.5 Shrnutí Barnesova-Hutova algoritmu

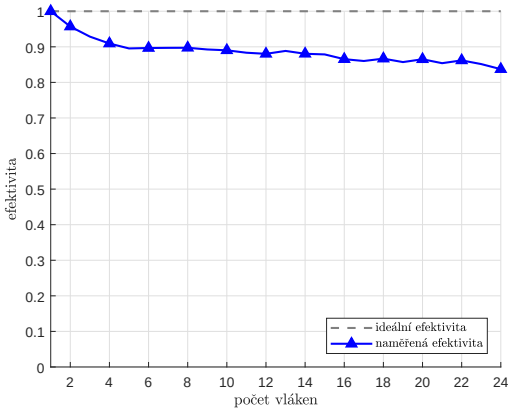
V tabulce 6.8 jsou shrnuty časy jednotlivých částí Barnesova-Hutova algoritmu a je spočítán jejich podíl na celkovém času pro různé počty vláken. Jsou zde také vypsány celkové časy běhu, zrychlení a efektivity celého procesu výpočtu sil Barnesovým-Hutovým algoritmem. Lze si všimnout, že ze všeho nejdéle trvá výpočet sil. Na jednom vlákně zabírá 95 % času, s rostoucím

Tabulka 6.7: Časy výpočtu sil Barnesovým-Hutovým algoritmem na různém počtu vláken

počet vláken	1	2	4	8	12	16	24
čas [ms]	12555.40	6559.50	3452.05	1748.99	1188.58	906.88	624.87
zrychlení	1	1.91	3.63	7.17	10.56	13.84	20.09
efektivita [%]	100	95.70	90.92	89.73	88.02	86.52	83.71



Obrázek 6.12: Závislost času výpočtu sil Barnesovým-Hutovým algoritmem na počtu vláken



Obrázek 6.13: Efektivita výpočtu sil Barnesovým-Hutovým algoritmem v závislosti na počtu vláken

Tabulka 6.8: Shrnutí časů výpočtu jednotlivých částí Barnesova-Hutova algoritmu

počet vláken	1	2	4	8	12	16	24
konstrukce [ms]	479.70	327.84	209.67	149.91	142.07	144.49	144.02
výp. vlastností [ms]	109.29	74.94	41.42	21.45	14.55	11.81	10.56
výpočet sil [ms]	12555.40	6559.50	3452.05	1748.99	1188.58	906.88	624.87
celkový čas [ms]	13144.39	6962.28	3703.14	1920.36	1345.21	1063.19	779.46
podíl konstrukce	0.0364	0.0470	0.0566	0.0780	0.1056	0.1359	0.1847
podíl výp. vlastností	0.0083	0.0107	0.0111	0.0111	0.0108	0.0111	0.0135
podíl výpočtu sil	0.9551	0.9421	0.9321	0.9107	0.8835	0.8529	0.8016
zrychlení	1	1.88	3.54	6.84	9.77	12.36	16.86
efektivita [%]	100	94.39	88.73	85.55	81.42	77.26	70.26

počtem vláken se snižuje až na 80 % na 24 vláknech, a to díky velmi dobré paralelní škálovatelnosti oproti ostatním částem. Kvůli špatné škálovatelnosti konstrukce stromu se její časový podíl s rostoucím počtem vláken zvyšuje. Ze 4 % na jednom vlákně vzroste na 18 % na 24 vláknech. Podíl výpočtu vlastností buněk je minimální, pohybuje se okolo 1 %. Celý algoritmus paralelně škáluje přijatelně, nejnižší efektivita je na 24 vláknech přibližně 70 %. Žádný zlom v efektivitě se v okolí 12 vláken neprojevil.

Poznamenejme, že ačkoli má na jednom vlákně výpočet vlastností buněk časový podíl méně než jedno procento, má smysl tento proces paralelizovat. Na 24 vláknech by totiž v opačném případě zabíral více než 10 % času a s rostoucím počtem vláken by se tento podíl zvyšoval. Z Amdahlova zákona plyne, že maximální paralelní zrychlení by v tomto případě bylo asi 120. Je však třeba se obávat spíše času konstrukce stromu, který je zespoda omezený přibližně 140 milisekundami. Na 12 vláknech zabírá asi 10 % času a na vyšším počtu vláken už neškáluje. Z Amdahlova zákona tedy lze odvodit, že nejvyšší možné zrychlení, kterého by bylo možné pro tento případ na libovolném počtu vláken docílit, je přibližně 90.

V tabulce jsou opět zobrazeny hodnoty z testování na systému $2^{20} \approx 10^6$ částic generovaných podle Plummerova modelu. Byla zvolena hodnota parametru $\theta = 1$, při nižších hodnotách trvá výpočet sil delší dobu, má tedy větší podíl na čase a algoritmus lépe škáluje. Při použití rovnoměrného rozložení částic v kouli trvala na 24 vláknech konstrukce stromu 28 % a výpočet sil 70 % času, zbylá dvě procenta náleží výpočtu vlastností buněk. Celková doba výpočtu sil byla oproti Plummerovu modelu přibližně poloviční. Efektivita klesla u 24 vláken na 60 %. Počet vláken byl pro konstrukci stromu omezen na maximální počet 12.

6.7 Porovnání časové náročnosti algoritmů

V kapitole 4 bylo zmíněno, že výpočet sil přímou metodou má časovou složitost $\mathcal{O}(n^2)$, zatímco Barnesův-Hutův algoritmus má složitost $\mathcal{O}(n \log n)$. Doba výpočtu sil tedy v závislosti na počtu částic roste kvadraticky, resp. lineárně-logaritmicky. To nám ale nic neřekne o konkrétním čase, který výpočet sil zabere – asymptotická notace totiž zanedbává konstanty.

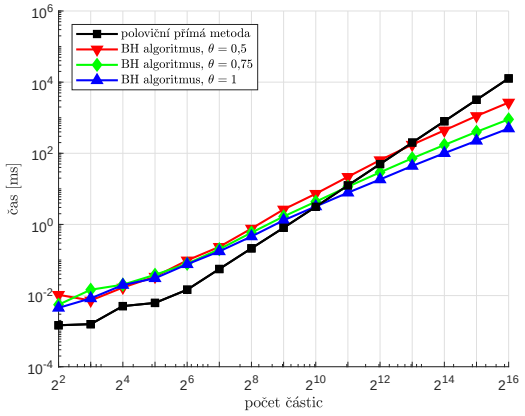
Z definice časové složitosti [8] plyne, že pro algoritmus s časovou složitostí $\mathcal{O}(g(n))$ lze pro dostatečně velká n shora odhadnout čas běhu tohoto algoritmu hodnotou $T(n) = Cg(n)$, kde $C > 0$ je konstanta daná implementací příslušného algoritmu a rychlostí výpočetního stroje. Pokud tuto konstantu nalezneme, budeme mít vzorec pro horní odhad doby běhu algoritmu v závislosti na velikosti vstupních dat n.

V této podkapitole se mimo jiné pokusíme nalézt konstantu C a tedy i odhad doby běhu pro oba algoritmy výpočtu sil. Konstantu budeme hledat tak, že pro několik vstupních souborů s různým počtem částic změříme dobu výpočtu sil. Pro dané měření spočítáme odhad konstanty vztahem $C = T(n)/g(n)$ a budeme sledovat, na které hodnotě se ustálí.

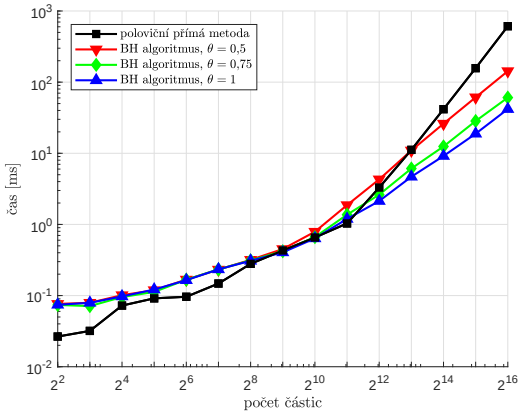
V tabulce 6.9 jsou vypsány časy výpočtu sil poloviční přímou metodou a Barnesovým-Hutovým algoritmem pro tři různé hodnoty parametru θ v závislosti na počtu částic v systému. Na obrázku 6.14 jsou tyto hodnoty vykresleny graficky. V tabulce jsou také spočítány odhady příslušných konstant. Výpočty provádíme na jednom vlákně na systému generovaném podle Plummerova modelu a používáme datový typ double. Na obrázku 6.15 jsou vykresleny časy výpočtu sil na 24 vláknech. U Barnesova-Hutova algoritmu měříme čas celého procesu výpočtu sil i s konstrukcí stromu a výpočtem vlastností buněk. Používáme logaritmus o základu 2.

Tabulka 6.9: Porovnání časů výpočtu sil obou algoritmů pro různé počty částic

počet částic	16	64	256	1024	4096	16384	65536
přímá metoda [ms]	0.00506	0.01461	0.21307	3.1402	49.764	796.09	12731.4
odhad C_P ($\times 10^{-6}$)	19.7650	3.5668	3.2512	2.9947	2.9661	2.9656	2.9642
BH, $\theta = 0{,}5$ [ms]	0.01683	0.09753	0.76491	7.2873	64.142	440.74	2673.74
odhad $C_{B,0,5}$ ($\times 10^{-6}$)	263.02	254.01	373.49	711.65	1304.98	1921.49	2549.87
BH, $\theta = 0{,}75$ [ms]	0.02024	0.07866	0.58754	4.3669	29.149	171.20	899.83
odhad $C_{B,0,75}$ ($\times 10^{-6}$)	316.37	204.84	286.88	426.45	593.04	746.37	858.14
BH, $\theta = 1$ [ms]	0.01980	0.07584	0.45907	3.2029	18.377	100.69	497.44
odhad $C_{B,1}$ ($\times 10^{-6}$)	309.51	197.52	224.15	312.78	373.88	438.97	474.40



Obrázek 6.14: Čas výpočtu sil pro různé počty částic v systému na 1 vlákně



Obrázek 6.15: Čas výpočtu sil pro různé počty částic v systému na 24 vláknech

Z tabulky lze usoudit, že konstantu poloviční přímé metody můžeme bezpečně odhadnout hodnotou $C_P = 3 \cdot 10^{-6}$. Pro systém s více než 1000 částicemi jsme tedy schopni vzorcem $T_P(n) = 3 \cdot 10^{-6} n^2$ odhadnout dobu výpočtu sil (v milisekundách) poloviční přímou metodou. Podobný experiment jsme provedli i pro úplnou přímou metodu a získali jsme pro ni odhad konstanty $5 \cdot 10^{-6}$.

U Barnesova-Hutova algoritmu lze vidět, že se čas výpočtu s vyšší hodnotou parametru θ snižuje. Konstanta se však u všech tří hodnot parametru θ s rostoucím počtem částic n zvyšuje. Pravděpodobně jsme tedy ještě nedosáhli dostatečně velkého n. Měření jsme pro Barnesův-Hutův algoritmus prováděli až do $2^{20} \approx 10^6$ částic. Pro tento počet částic nabývaly odhady konstant hodnot $C_{B,0,5} = 4{,}1 \cdot 10^{-3}$, $C_{B,0,75} = 1{,}3 \cdot 10^{-3}$, resp. $C_{B,1} = 0{,}63 \cdot 10^{-3}$.

Můžeme porovnat, jak dlouho by každým algoritmem trvalo na jednom vlákně vypočítat vzájemné síly působící mezi $2^{20} \approx 10^6$ částicemi. Pro Barnesův-Hutův algoritmus s volbou $\theta = 0{,}5$ jsme tuto hodnotu změřili – výpočet sil trval 87 vteřin. Pro odhad doby výpočtu sil přímou metodou použijeme dříve uvedený vzorec. Dosazením do něj dostaneme, že by výpočet sil trval necelou hodinu.

Z tabulky a grafů lze usoudit, že výpočet sil Barnesovým-Hutovým algoritmem je pro velké počty částic v systému dle očekávání rychlejší než použití přímé metody. Pro malé počty částic se však časově vyplatí provést simulaci přímou metodou. Barnesův-Hutův algoritmus je výhodnější použít pro $\theta = 1$ až od 1000 částic, pro $\theta = 0{,}75$ se tato hodnota posune na 2000 částic, u $\theta = 0{,}5$ se hranice navýší až na 6500 částic v systému. Z důvodu horší paralelní škálovatelnosti Barnesova-Hutova algoritmu oproti přímé metodě se kritické hodnoty na vyšším počtu vláken posunou ještě výše, viz obrázek 6.15.

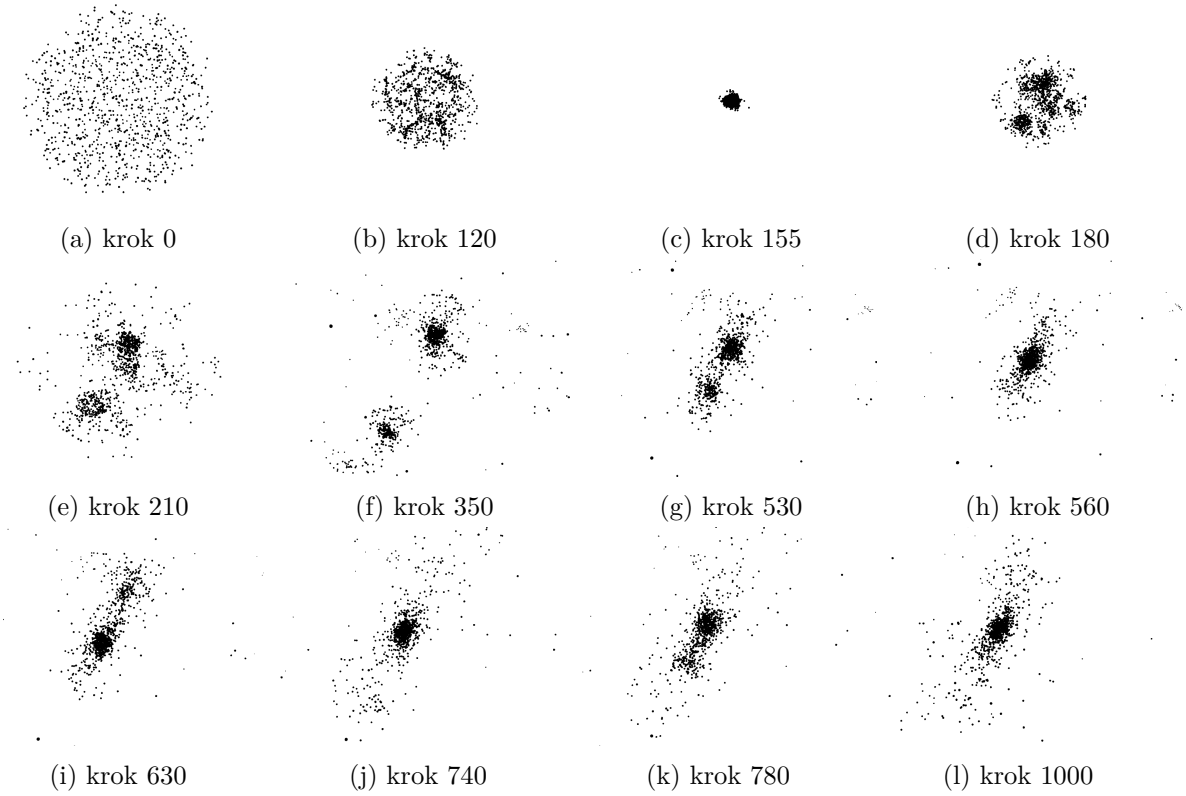
6.8 Výstup simulací

Pomocí vytvořeným programu jsme provedli několik simulací hvězdných systémů, jejichž výstupy jsou v této kapitole vizualizovány a popsány. Všechny simulace byly prováděny s použitím Rungeho-Kuttovy metody čtvrtého řádu a pro výpočet sil byl použit Barnesův-Hutův algoritmus s parametrem $\theta = 0{,}5$. Animace provedených simulací jsou k dispozici v příloze této práce.

6.8.1 Rovnoměrné rozložení

Na sérii obrázků 6.16 je zobrazen grafický výstup simulace systému s počátečním rovnoměrným rozložením 1024 částic v kouli s nulovou počáteční rychlostí. Zvolili jsme celkovou hmotnost $M = 10^{35}$ kg $\approx 5 \cdot 10^4 M_\odot$ a poloměr koule obsahující částice $a = 10^{16}$ m $\approx 0{,}33$ pc. Na základě těchto vlastností jsme spočetli hodnotu zjemňovacího parametru $\epsilon = 10^{15}$ m a zvolili jsme délku časového kroku $3 \cdot 10^9$ s. Simulovali jsme 1000 časových kroků, celkový simulační čas je tedy roven $3 \cdot 10^{12}$ s. Celá simulace trvala na 24 vláknech přibližně tři vteřiny.

V prvním kroku se částice nacházejí v poloze dané vygenerovanými počátečními podmínkami. Protože částice mají z počátku nulovou rychlost, začnou všechny směřovat směrem ke společnému těžišti. Začnou se projevovat mírné fluktuace v hustotě částic a ve 155. kroku se částice koncentrují uprostřed systému. Částice mají v tu dobu vysokou rychlost, začnou se od sebe tedy opět vzdalovat. Počáteční fluktuace v hustotě začínají být podstatně znát a zformují se dva větší shluky částic. Ty se od sebe drží delší dobu daleko, nakonec ale jejich vzájemná gravitace způsobí jejich kolizi. Zbytky menšího shluku pak s větším shlukem zkolidují ještě jednou.



(a) krok 0 (b) krok 120 (c) krok 155 (d) krok 180

(e) krok 210 (f) krok 350 (g) krok 530 (h) krok 560

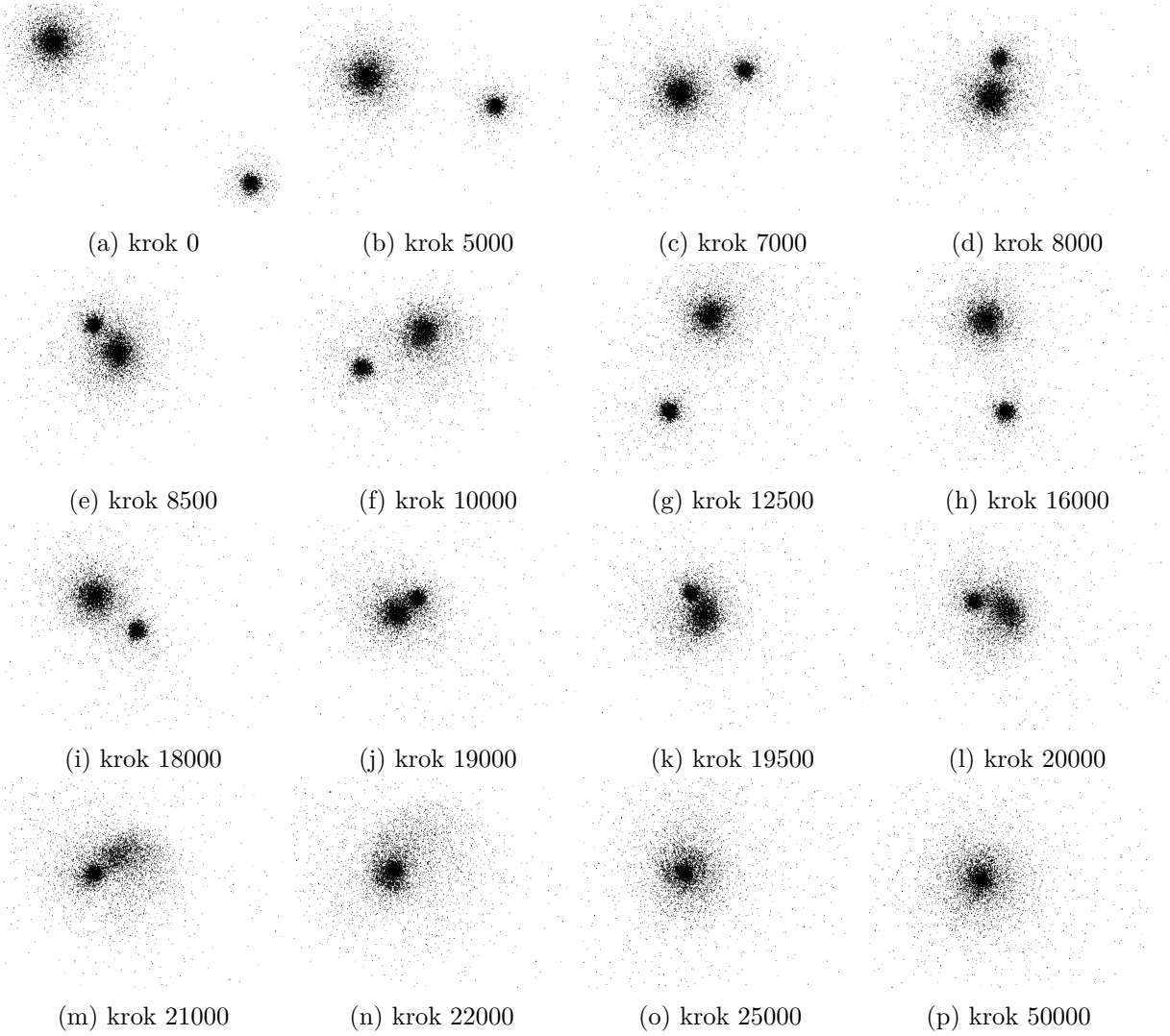
(i) krok 630 (j) krok 740 (k) krok 780 (l) krok 1000

Obrázek 6.16: Simulace systému s počátečním rovnoměrným rozložením 1024 částic v kouli s nulovou počáteční rychlostí

6.8.2 Kolize dvou hvězdokup

Na sérii obrázků 6.17 je graficky zobrazen výstup simulace systému, v němž se zpočátku nacházejí dvě hvězdokupy generované podle Plummerova modelu. Větší z nich obsahuje 5000 částic o celkové hmotnosti $5 \cdot 10^{35}$ kg $\approx 2{,}5 \cdot 10^5 M_\odot$ a poloměru jádra $5 \cdot 10^{16}$ m $\approx 1{,}66$ pc. Menší hvězdokupa má všechny zmíněné parametry dvoupětinové. Hodnotu zjemňovacího parametru jsme na základě vlastností systému vyhodnotili na $\epsilon = 2 \cdot 10^{15}$ m a zvolili jsme délku časového kroku $\Delta t = 10^{10}$ s. Simulovali jsme celkem 50000 kroků, celkový simulační čas je tedy $5 \cdot 10^{14}$ s (přibližně 15 milionů let). Celá simulace na 24 vláknech trvala přibližně půl hodiny.

Hvězdokupy se k sobě zpočátku přibližují se vzrůstající rychlostí, a to díky jejich vzájemnému gravitačnímu působení. První vzájemná blízká interakce okolo kroku 8500 proběhne bez kolize. Při druhé blízké interakci mezi kroky 19000 a 21000 už ale hvězdokupy zkolidují. Menší hvězdokupa způsobí způsobí rozpad té větší, načež její částice prakticky pohltí, čímž vznikne jediná velká hvězdokupa. Na videu v příloze je pak dobře vidět rotace celého systému proti směru hodinových ručiček.



(a) krok 0 (b) krok 5000 (c) krok 7000 (d) krok 8000

(e) krok 8500 (f) krok 10000 (g) krok 12500 (h) krok 16000

(i) krok 18000 (j) krok 19000 (k) krok 19500 (l) krok 20000

(m) krok 21000 (n) krok 22000 (o) krok 25000 (p) krok 50000

Obrázek 6.17: Simulace kolize dvou hvězdokup generovaných podle Plummerova modelu o celkovém počtu 7000 částic

7 Závěr

V této práci jsme se zabývali problémem mnoha těles a jeho řešením. Probrali jsme několik metod numerického řešení počátečních úloh, které jsou nezbytnou součástí řešení tohoto problému. Zabývali jsme se Eulerovou metodou, metodou leapfrog a Rungeho-Kuttovými metodami. Vybrané numerické metody jsme posléze aplikovali na problém mnoha těles.

Dále jsme se zaměřili na dva algoritmy výpočtu sil působících na částice – přímou metodu s časovou složitostí $\mathcal{O}(n^2)$ a zejména na Barnesův-Hutův algoritmus, který má příznivější časovou složitost $\mathcal{O}(n \log n)$. U přímě metody jsme zmínili její úplnou a poloviční variantu. Pro Barnesův-Hutův algoritmus jsme pak představili dvě metody konstrukce stromu – variantu vkládání a dělení.

Program pro řešení problému mnoha těles jsme implementovali v jazyce C++. Výpočty jsme optimalizovali a paralelizovali jsme je pomocí standardu OpenMP, kterému jsme se věnovali podrobněji. V práci jsme také probrali strukturu vytvořeného programu a jeho použití.

Na výpočetním uzlu clusteru Salomon jsme provedli několik numerických experimentů, kterými jsme otestovali efektivitu vytvořeného programu. Porovnali jsme jednotlivé varianty algoritmů pro výpočet sil a otestovali jsme jejich paralelní škálovatelnost. Ukázalo se, že poloviční varianta přímé metody se skutečně časově vyplatí a škáluje téměř ideálně. Konstrukce stromu Barnesova-Hutova algoritmu probíhala nejrychleji pomocí řadícího dělení, v okolí 11 vláken však bylo časově výhodnější indexovací dělení. Při výpočtech na 13 a více vláknech docházelo k tzv. NUMA efektu, čímž se doba konstrukce stromu výrazně zvýšila. Konstrukce stromu neškálovala dobře ani na nízkém počtu vláken. Výpočet vlastností buněk však škáloval přijatelně a u samotného výpočtu sil jsme dosáhli téměř ideální škálovatelnosti až do celkového počtu 24 vláken.

Dále jsme otestovali dopad některých provedených optimalizací. Zjistili jsme, že uspořádání paměti stylem SoA je skutečně časově výhodnější. Použití vektorizace zrychlilo výpočty až na jednu výjimku dle očekávání. Provedli jsme také několik numerických experimentů, kterými jsme porovnali řád numerických metod pro řešení počátečních úloh a zjistili jsme přesnost výpočtu sil pomocí Barnesova-Hutova algoritmu. Výsledky těchto experimentů odpovídaly očekáváním. Nakonec jsme výstupy některých simulací vizualizovali.

Na tuto práci by mohlo navázat studium a implementace sofistikovanějších metod, mezi něž patří například algoritmus FMM (fast multipole method). Tato metoda umožňuje dosáhnout až lineární časové složitosti a lze ji využít nejen k částicovým simulacím, ale také k urychlení řešení parciálních diferenciálních rovnic pomocí metody hraničních prvků.

A Přílohy práce

V elektronické příloze se nacházejí zdrojové kódy vytvořených programů, skripty pro jejich překlad a testování, výstupní data z měření na uzlu clusteru Salomon a vizualizace některých simulací. Adresářová struktura přílohy je následující:

_Animations: videa, animace, grafické výstupy simulací

_Data: vstupní a výstupní soubory simulací

_ResultsSalomon: výstupy měření času a chyb numerických metod, které byly měřeny na výpočetním uzlu clusteru Salomon

_Scripts: skripty používané pro simulace, měření, kompilaci programů aj.

ostatní složky: zdrojové kódy příslušného programu

Reference

[1] Barnes, J., and Hut, P. A hierarchical O(N log N) force-calculation algorithm. *Nature 324*, 6096 (1986), 446.

[2] Graps, A. N-body/particle simulation methods. Dostupné z `http://www.cs.cmu.edu/afs/cs/academic/class/15850c-s96/www/nbody.html`, 1996. Navštíveno 13. 2. 2019.

[3] Hager, G., and Wellein, G. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[4] Hayes, W. B. A brief survey of issues relating to the reliability of simulation of the large gravitational n-body problem. Dostupné z `http://www.cs.toronto.edu/~wayne/research/thesis/depth/depth.html`, 1996. Navštíveno 13. 2. 2019.

[5] Hut, P., and Makino, J. Initial Conditions: Plummer's model. Dostupné z `http://www.artcompsci.org/kali/vol/plummer/title.html`, 2007. Navštíveno 30. 3. 2019.

[6] Hut, P., McMillan, S., Makino, J., and Zwart, S. P. Starlab: A Software Environment for Collisional Stellar Dynamics. Dostupné z `https://www.sns.ias.edu/~starlab/`. Navštíveno 28. 4. 2019.

[7] James, J. D. M. Celestial mechanics notes, set 1: Introduction to the n-body problem. Dostupné z `http://cosweb1.fau.edu/~jmirelesjames/introductionNotes.pdf`, 2007. Navštíveno 12. 4. 2019.

[8] Jančar, P. Úvod do teoretické informatiky – učební text. Dostupné z `http://www.cs.vsb.cz/sawa/uti/materialy/uti.pdf`, 2007. Navštíveno 25. 4. 2019.

[9] Janes, K. Star clusters. Dostupné z `http://www.astro.caltech.edu/~george/ay20/eaa-starclus.pdf`, 2001. Navštíveno 27. 4. 2019.

[10] Jeffers, J., Reinders, J., and Sodani, A. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier, 2016.

[11] Kitware, Inc. VTK File Formats. Dostupné z `https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf`. Navštíveno 28. 3. 2019.

[12] Krajc, B., and Beremlijski, P. *Obyčejné diferenciální rovnice*. Vysoká škola báňská-Technická univerzita, Ostrava, 2012.

[13] Lindholm, T. N-body algorithms. Dostupné z `http://www.cs.hut.fi/~ctl/NBody.pdf`, 1999. Navštíveno 13. 2. 2019.

[14] Mapelli, M. N-body techniques for astrophysics: Lecture 3 — N-body methods for collisionless systems. Dostupné z `http://web.pd.astro.it/mapelli/Nbody3.pdf`, 2015. Navštíveno 19. 2. 2019.

[15] Přikryl, P. *Numerické metody matematické analýzy*. SNTL, Praha, 1988.

[16] Prins, F. J. COMP 633 – Parallel Computing – OpenMP Case Study: The Barnes-Hut N-body Algorithm. Dostupné z https://www.cs.unc.edu/~prins/Classes/633/Slides/09-smm4.pdf, 2017. Navštíveno 19. 2. 2019.

[17] Rodionov, S. A., and Sotnikova, N. Y. Optimal choice of the softening length and time step in n-body simulations. *Astronomy Reports 49*, 6 (2005), 470–476.

[18] Terboven, C., and Schmidl, D. OpenMP Tasking. Dostupné z http://prace.it4i.cz/sites/prace.it4i.cz/files/files/advancedopenmptutorial_3.pdf. Navštíveno 24. 3. 2019.

[19] Trenti, M., and Hut, P. Gravitational N-body simulations. *Scholarpedia* (2008).

[20] Vondrák, V., and Pospíšil, L. *Numerické metody I.* Vysoká škola báňská-Technická univerzita, Ostrava, 2011.

[21] Young, P. The leapfrog method and other symplectic algorithms for integrating Newton's laws of motion. *Physics 115*, 242 (2014).

[22] Zwart, S. P. AMUSE, Astrophysical Multipurpose Software Environment. Dostupné z http://amusecode.org/. Navštíveno 28. 4. 2019.